Abortable Linearizable Modules

Rachid Guerraoui   Viktor Kuncak   Giuliano Losa

April 10, 2026

Abstract

We define the SLin I/O-automaton and prove its composition property. The SLin I/O-automaton is at the heart of the Speculative Linearizability framework. This framework simplifies devising robust distributed algorithms by enabling their decomposition into independent modules that can be analyzed and proved correct in isolation. It is particularly useful when working in a distributed environment, where the need to tolerate faults and asynchrony has made current monolithic protocols so intricate that it is no longer tractable to check their correctness. Our theory contains a formalization of simulation proof techniques in the I/O-automata of Lynch and Tuttle and a typical example of a refinement proof.

Contents

1 Introduction

Linearizability [2] is a key design methodology for reasoning about implementations of concurrent abstract data types in both shared memory and message passing systems. It presents the illusion that operations execute sequentially and fault-free, despite the asynchrony and faults that are often present in a concurrent system, especially a distributed one.

However, devising complete linearizable objects is very difficult, especially in the presence of process crashes and asynchrony, requiring complex algorithms (such as Paxos [3]) to work correctly under general circumstances, and often resulting in bad average-case behavior. Concurrent algorithm designers therefore resort to speculation, i.e. to optimizing existing algorithms to handle common scenarios more efficiently. More precisely, a speculative systems has a fall-back mode that works in all situations and several optimization modes, each of which is very efficient in a particular situation but might not work at all in some other situation. By observing its execution, a speculative system speculates about which particular situation it will be subject to and chooses the most efficient mode for that situation. If speculation reveals wrong, a new speculation is made in light of newly available observations. Unfortunately, building speculative system ad-hoc results in protocols so complex that it is no longer tractable to prove their correctness.

We the specification of the SLin (a shorthand for Speculative Linearizability) I/O-automaton [5], which can be used to build a speculatively linearizable algorithm out of independent modules that each implement the different modes of the speculative algorithm. The SLin I/O-automaton is at the heart of the Speculative Linearizability framework [4, 1]. The Speculative Linearizability framework first appeared in [1] and was later refined in [4]. This development is based on the later [4].

The SLin I/O-automaton produces traces that are linearizable with respect to a generic type of object. Moreover, the composition of two instances of the SLin I/O-automaton behaves like a single instance. Hence it is guaranteed that the composition of any number of instances of the SLin I/O-automaton is linearizable. In this formal development, we prove the idempotence theorem, i.e. that the composition of two instances of the SLin I/O-automaton is itself an implementation of the SLin I/O-automaton.

The properties stated above simplify the development and analysis of speculative systems: Instead of having to reason about an entanglement of complex protocols, one can devise several modules with the property that, when taken in isolation, each module refines the SLin I/O-automaton. Hence complex protocols can be divided into smaller modules that can be analyzed independently of each other. In particular, it allows to optimize an existing protocol by creating separate optimization modules, prove each optimization correct in isolation, and obtain the correctness of the overall protocol from the correctness of the existing one.

In this document we define the SLin I/O-automaton and prove the Composition Theorem, which states that the composition of two instances of the SLin I/O-automaton behaves as a single instance of the SLin I/O-automaton. We use a refinement mapping to establish this fact.

2 Sequences as Lists

```
theory Sequences
imports Main
begin

locale Sequences
begin
```

We reverse the order of application of (#) and (@) because it we think that it is easier to think of sequences as growing to the right.

```
no-notation Cons (infixr ‹#› 65)
abbreviation Append  (infixl ‹#› 65)
  where Append xs x ≡ Cons x xs
no-notation append (infixr ‹@› 65)
abbreviation Concat  (infixl ‹@› 65)
  where Concat xs ys ≡ append ys xs

end

end
```

3 I/O Automata with Finite-Trace Semantics

```
theory IOA
imports Main Sequences
begin
```

This theory is inspired and draws material from the IOA theory of Nipkow and Müller

```
locale IOA = Sequences

record 'a signature =
  inputs::'a set
  outputs::'a set
  internals::'a set

context IOA
begin
```

3.1 Signatures

```
definition actions :: 'a signature ⇒ 'a set where
  actions asig ≡ inputs asig ∪ outputs asig ∪ internals asig

definition externals :: 'a signature ⇒ 'a set where
  externals asig ≡ inputs asig ∪ outputs asig

definition locals :: 'a signature ⇒ 'a set where
  locals asig ≡ internals asig ∪ outputs asig

definition is-asig :: 'a signature ⇒ bool where
  is-asig triple ≡
     inputs triple ∩ outputs triple = {} ∧
     outputs triple ∩ internals triple = {} ∧
     inputs triple ∩ internals triple = {}

lemma internal-inter-external:
  assumes is-asig sig
  shows internals sig ∩ externals sig = {}
  using assms by (auto simp add:internals-def externals-def is-asig-def)

definition hide-asig where
  hide-asig asig actns ≡
    (|inputs = inputs asig − actns, outputs = outputs asig − actns,
      internals = internals asig ∪actns|)

end
```

3.2 I/O Automata

```
type-synonym
```

(*'s*,*'a*) *transition* = *'s* × *'a* × *'s*

record (*'s*,*'a*) *ioa* =
asig::*'a signature*
start::*'s set*
trans::(*'s*,*'a*)*transition set*

context *IOA*
begin

abbreviation *act A* ≡ *actions* (*asig A*)
abbreviation *ext A* ≡ *externals* (*asig A*)
abbreviation *int* **where** *int A* ≡ *internals* (*asig A*)
abbreviation *inp A* ≡ *inputs* (*asig A*)
abbreviation *out A* ≡ *outputs* (*asig A*)
abbreviation *local A* ≡ *locals* (*asig A*)

definition *is-ioa*::(*'s*,*'a*) *ioa* ⇒ *bool* **where**
is-ioa A ≡ *is-asig* (*asig A*)
∧ (∀ *triple* ∈ *trans A* . (*fst o snd*) *triple* ∈ *act A*)

definition *hide* **where**
hide A actns ≡ *A*⦇*asig* := *hide-asig* (*asig A*) *actns*⦈

definition *is-trans*::*'s* ⇒ *'a* ⇒ (*'s*,*'a*)*ioa* ⇒ *'s* ⇒ *bool* **where**
is-trans s1 a A s2 ≡ (*s1*,*a*,*s2*) ∈ *trans A*

notation
is-trans (‹- —-—-⟶ -› [*81*,*81*,*81*,*81*] *100*)

definition *rename-set* **where**
rename-set A ren ≡ {*b*. ∃ *x* ∈ *A* . *ren b* = *Some x*}

definition *rename* **where**
rename A ren ≡
⦇*asig* = ⦇*inputs* = *rename-set* (*inp A*) *ren*,
outputs = *rename-set* (*out A*) *ren*,
internals = *rename-set* (*int A*) *ren*⦈,
start = *start A*,
trans = {*tr*. ∃ *x* . *ren* (*fst* (*snd tr*)) = *Some x* ∧ (*fst tr*) −*x*−*A*⟶ (*snd* (*snd tr*))}⦈

Reachable states and invariants

inductive
reachable :: (*'s*,*'a*) *ioa* ⇒ *'s* ⇒ *bool*
for *A* :: (*'s*,*'a*) *ioa*
where
reachable-0: *s* ∈ *start A* ⟹ *reachable A s*
| *reachable-n*: ⟦ *reachable A s*; *s* −*a*−*A*⟶ *t* ⟧ ⟹ *reachable A t*

```
definition invariant where
  invariant A P ≡ (∀ s . reachable A s ⟶ P(s))

theorem invariantI:
  fixes A P
  assumes ⋀ s . s ∈ start A ⟹ P s
  and ⋀ s t a . ⟦reachable A s; P s; s −a−A⟶ t⟧ ⟹ P t
  shows invariant A P
proof −
  { fix s
    assume reachable A s
    hence P s
    proof (induct rule:reachable.induct)
      fix s
      assume s ∈ start A
      thus P s using assms(1) by simp
    next
      fix a s t
      assume reachable A s and P s and  s −a−A⟶ t
      thus P t using assms(2) by simp
    qed }
  thus ?thesis by (simp add:invariant-def)
qed

end
```

3.3 Composition of Families of I/O Automata

```
record ('id, 'a) family =
  ids :: 'id set
  memb :: 'id ⇒ 'a

context IOA
begin

definition is-ioa-fam where
  is-ioa-fam fam ≡ ∀ i ∈ ids fam . is-ioa (memb fam i)

definition compatible2  where
  compatible2 A B ≡
   out A ∩ out B = {} ∧
   int A ∩ act B = {} ∧
   int B ∩ act A = {}

definition compatible::('id, ('s,'a)ioa) family ⇒ bool where
  compatible fam ≡ finite (ids fam) ∧
    (∀ i ∈ ids fam . ∀ j ∈ ids fam . i ≠ j ⟶
      compatible2 (memb fam i) (memb fam j))
```

```
definition asig-comp2 where
  asig-comp2 A B ≡
     ⦇inputs = (inputs A ∪ inputs B) − (outputs A ∪ outputs B),
      outputs = outputs A ∪ outputs B,
      internals = internals A ∪ internals B⦈

definition asig-comp::('id, ('s,'a)ioa) family ⇒ 'a signature where
  asig-comp fam ≡
    ⦇ inputs = ⋃i∈(ids fam). inp (memb fam i)
        − (⋃i∈(ids fam). out (memb fam i)),
      outputs = ⋃i∈(ids fam). out (memb fam i),
      internals = ⋃i∈(ids fam). int (memb fam i) ⦈

definition par2 (infixr ‹∥› 10) where
  A ∥ B ≡
      ⦇asig = asig-comp2 (asig A) (asig B),
       start = {pr. fst pr ∈ start A ∧ snd pr ∈ start B},
       trans = {tr.
        let s = fst tr; a = fst (snd tr); t = snd (snd tr)
        in (a ∈ act A ∨ a ∈ act B)
           ∧ (if a ∈ act A
              then fst s −a−A⟶ fst t
              else fst s = fst t)
           ∧ (if a ∈ act B
              then snd s −a−B⟶ snd t
              else snd s = snd t) }⦈

definition par::('id, ('s,'a)ioa) family ⇒ ('id ⇒ 's,'a)ioa where
  par fam ≡ let ids = ids fam; memb = memb fam in
      ⦇ asig = asig-comp fam,
        start = {s . ∀ i∈ids . s i ∈ start (memb i)},
        trans = { (s, a, s') .
          (∃ i∈ids . a ∈ act (memb i))
         ∧ (∀ i∈ids .
              if a ∈ act (memb i)
              then s i −a−(memb i)⟶ s' i
              else s i = (s' i)) } ⦈

lemmas asig-simps = hide-asig-def is-asig-def locals-def externals-def actions-def
  hide-def compatible-def asig-comp-def
lemmas ioa-simps = rename-def rename-set-def is-trans-def is-ioa-def par-def

end
```

3.4 Executions and Traces

```
type-synonym
  ('s,'a)pairs = ('a×'s) list
```

```
type-synonym
  ('s,'a)execution = 's × ('s,'a)pairs
type-synonym
  'a trace = 'a list

record ('s,'a)execution-module =
  execs::('s,'a)execution set
  asig::'a signature

record 'a trace-module =
  traces::'a trace set
  asig::'a signature

context IOA
begin

fun is-exec-frag-of::('s,'a)ioa ⇒ ('s,'a)execution ⇒ bool where
  is-exec-frag-of A (s,(ps#p')#p) =
    (snd p' −fst p−A⟶ snd p ∧ is-exec-frag-of A (s, (ps#p')))
| is-exec-frag-of A (s, [p]) = s −fst p−A⟶ snd p
| is-exec-frag-of A (s, []) = True

definition is-exec-of::('s,'a)ioa ⇒ ('s,'a)execution ⇒ bool where
  is-exec-of A e ≡ fst e ∈ start A ∧ is-exec-frag-of A e

definition filter-act where
  filter-act ≡ map fst

definition schedule where
  schedule ≡ filter-act o snd

definition trace where
  trace sig ≡ filter (λ a . a ∈ externals sig) o schedule

definition is-schedule-of where
  is-schedule-of A sch ≡
    (∃ e . is-exec-of A e ∧ sch = filter-act (snd e))

definition is-trace-of where
  is-trace-of A tr ≡
    (∃ sch . is-schedule-of A sch ∧ tr = filter (λ a. a ∈ ext A) sch)

definition traces where
  traces A ≡ {tr. is-trace-of A tr}

lemma traces-alt:
  shows traces A = {tr . ∃ e . is-exec-of A e
    ∧ tr = trace (ioa.asig A) e}
proof −
```

```
 { fix t
   assume a:t ∈ traces A
   have ∃ e . is-exec-of A e ∧ trace (ioa.asig A) e = t
   proof −
     from a obtain sch where 1:is-schedule-of A sch
       and 2:t = filter (λ a. a ∈ ext A) sch
       by (auto simp add:traces-def is-trace-of-def)
     from 1 obtain e where 3:is-exec-of A e and 4:sch = filter-act (snd e)
       by (auto simp add:is-schedule-of-def)
     from 4 and 2 have trace (ioa.asig A) e = t
       by (simp add:trace-def schedule-def)
     with 3 show ?thesis by fast
   qed }
 moreover
 { fix e
   assume is-exec-of A e
   hence trace (ioa.asig A) e ∈ traces A
     by (force simp add:trace-def schedule-def traces-def
         is-trace-of-def is-schedule-of-def is-exec-of-def) }
 ultimately show ?thesis by blast
qed

lemmas trace-simps = traces-def is-trace-of-def is-schedule-of-def filter-act-def is-exec-of-def
 trace-def schedule-def

definition proj-trace::'a trace ⇒ ('a signature) ⇒ 'a trace (infixr ‹|› 12) where
 proj-trace t sig ≡ filter (λ a . a ∈ actions sig) t

definition ioa-implements :: ('s1,'a)ioa ⇒ ('s2,'a)ioa ⇒ bool   (infixr ‹=<|› 12)
where
 A =<| B ≡ inp A = inp B ∧ out A = out B ∧ traces A ⊆ traces B
```

3.5 Operations on Executions

```
definition cons-exec where
 cons-exec e p ≡ (fst e, (snd e)#p)

definition append-exec where
 append-exec e e' ≡ (fst e, (snd e)@(snd e'))

fun last-state where
 last-state (s,[]) = s
| last-state (s,ps#p) = snd p

lemma last-state-reachable:
 fixes A e
 assumes is-exec-of A e
 shows reachable A (last-state e) using assms
proof −
```

```
  have is-exec-of A e ⟹ reachable A (last-state e)
  proof (induction snd e arbitrary: e)
    case Nil
    from Nil.prems have 1:fst e ∈ start A by (simp add:is-exec-of-def)
    from Nil.hyps have 2:last-state e = fst e by (metis last-state.simps(1) surjective-pairing)
    from 1 and 2 and Nil.hyps show ?case by (metis reachable-0)
  next
    case (Cons p ps e)
    let ?e′ = (fst e, ps)
    have ih:reachable A (last-state ?e′)
    proof −
      from Cons.prems and Cons.hyps(2) have is-exec-of A ?e′
        by (simp add:is-exec-of-def)
       (metis (full-types) IOA.is-exec-frag-of.simps(1) IOA.is-exec-frag-of.simps(3)

           neq-Nil-conv prod.collapse)
      with Cons.hyps(1) show ?thesis by auto
    qed
     from Cons.prems and Cons.hyps(2) have (last-state ?e′)−(fst p)−A⟶(snd p)
       by (simp add:is-exec-of-def) (cases (A,fst e,ps#p) rule:is-exec-frag-of.cases, auto)
    with ih and Cons.hyps(2) show ?case
      by (metis last-state.simps(2) reachable.simps surjective-pairing)
  qed
  thus ?thesis using assms by fastforce
qed

lemma trans-from-last-state:
  assumes is-exec-frag-of A e and (last-state e)−a−A⟶s′
  shows is-exec-frag-of A (cons-exec e (a,s′))
    using assms by (cases (A, fst e, snd e) rule:is-exec-frag-of.cases, auto simp add:cons-exec-def)

lemma exec-frag-prefix:
  fixes A p ps
  assumes is-exec-frag-of A (cons-exec e p)
  shows is-exec-frag-of A e
    using assms by (cases (A, fst e, snd e) rule:is-exec-frag-of.cases, auto simp add:cons-exec-def)

lemma trace-same-ext:
  fixes A B e
  assumes ext A = ext B
  shows trace (ioa.asig A) e = trace (ioa.asig B) e
  using assms by (auto simp add:trace-def)

lemma trace-append-is-append-trace:
```

```
  fixes e e′ sig
  shows trace sig (append-exec e′ e) = trace sig e′ @ trace sig e
  by (simp add:append-exec-def trace-def schedule-def filter-act-def)

lemma append-exec-frags-is-exec-frag:
  fixes e e′ A as
  assumes is-exec-frag-of A e and last-state e = fst e′
  and is-exec-frag-of A e′
  shows is-exec-frag-of A (append-exec e e′)
proof −
  from assms show ?thesis
  proof (induct (fst e′,snd e′) arbitrary:e′ rule:is-exec-frag-of.induct)
    case (3 A)
    from 3.hyps and 3.prems(1)
    show ?case by (simp add:append-exec-def)
  next
    case (2 A p)
    have last-state e −(fst p)−A⟶ snd p using 2.prems(2,3) and 2.hyps
      by (metis is-exec-frag-of.simps(2) prod.collapse)
    hence is-exec-frag-of A (fst e, (snd e)#p) using 2.prems(1)
      by (metis cons-exec-def prod.collapse trans-from-last-state)
    moreover
    have append-exec e e′ = (fst e, (snd e)#p) using 2.hyps
      by (metis append-Cons append-Nil append-exec-def)
    ultimately
    show ?case by auto
  next
    case (1 A ps p′ p e′)
    have is-exec-frag-of A (fst e, (snd e)@((ps#p′)#p))
    proof −
      have is-exec-frag-of A (fst e, (snd e)@(ps#p′))
        by (metis 1.hyps 1.prems append-exec-def cons-exec-def
            exec-frag-prefix fst-conv prod-eqI snd-conv)
      moreover
      have snd p′ −(fst p)−A⟶ snd p using 1.prems(3) 1.hyps(2)
        by (metis is-exec-frag-of.simps(1) prod.collapse)
      ultimately show ?thesis by simp
    qed
    moreover have append-exec e e′ = (fst e, (snd e)@((ps#p′)#p))
      by (metis 1.hyps(2) append-exec-def)
    ultimately show ?case by simp
  qed
qed

lemma last-state-of-append:
  fixes e e′
  assumes fst e′ = last-state e
  shows last-state (append-exec e e′) = last-state e′
  using assms by (cases e′ rule:last-state.cases, auto simp add:append-exec-def)
```

end

end

4 Recoverable Data Types

theory *RDR*
imports *Main Sequences*
begin

4.1 The pre-RDR locale contains definitions later used in the RDR locale to state the properties of RDRs

locale *pre-RDR* = *Sequences* +
  fixes *δ*::*'a* ⇒ (*'b* × *'c*) ⇒ *'a* (**infix** ‹•› *65*)
  and *γ*::*'a* ⇒ (*'b* × *'c*) ⇒ *'d*
  and *bot*::*'a* (‹⊥›)
begin

fun *exec*::*'a* ⇒ (*'b*×*'c*)*list* ⇒ *'a* (**infix** ‹⋆› *65*) **where**
  exec s Nil = *s*
| *exec s* (*rs*#*r*) = (*exec s rs*) • *r*

definition *less-eq* (**infix** ‹⪯› *50*) **where**
  less-eq s s' ≡ ∃ *rs* . *s'* = (*s*⋆*rs*)

definition *less* (**infix** ‹≺› *50*) **where**
  less s s' ≡ *less-eq s s'* ∧ *s* ≠ *s'*

definition *is-lb* **where**
  is-lb s s1 s2 ≡ *s* ⪯ *s2* ∧ *s* ⪯ *s1*

definition *is-glb* **where**
  is-glb s s1 s2 ≡ *is-lb s s1 s2* ∧ (∀ *s'* . *is-lb s' s1 s2* ⟶ *s'* ⪯ *s*)

definition *contains* **where**
  contains s r ≡ ∃ *rs* . *r* ∈ *set rs* ∧ *s* = (⊥ ⋆ *rs*)

definition *inf* (**infix** ‹⊓› *65*) **where**
  inf s1 s2 ≡ *THE s* . *is-glb s s1 s2*

4.2 Useful Lemmas in the pre-RDR locale

lemma *exec-cons*:
  s ⋆ (*rs* # *r*)= (*s* ⋆ *rs*) • *r* **by** *simp*

lemma *exec-append*:
  (*s* ⋆ *rs*) ⋆ *rs'* = *s* ⋆ (*rs*@*rs'*)

```
proof (induct rs')
  show (s ⋆ rs) ⋆ [] = s ⋆ (rs@[]) by simp
next
  fix rs' r
  assume ih:(s ⋆ rs) ⋆ rs' = s ⋆ (rs@rs')
  thus (s ⋆ rs) ⋆ (rs'#r) = s ⋆ (rs @ (rs'#r))
    by (metis append-Cons exec-cons)
qed

lemma trans:
  assumes s1 ≼ s2 and s2 ≼ s3
  shows s1 ≼ s3 using assms
    by (auto simp add:less-eq-def, metis exec-append)

lemma contains-star:
  fixes s r rs
  assumes contains s r
  shows contains (s ⋆ rs) r
proof (induct rs)
  case Nil
  show contains (s ⋆ []) r using assms by auto
next
  case (Cons r' rs)
  with this obtain rs' where 1:s ⋆ rs = ⊥ ⋆ rs' and 2:r ∈ set rs'
    by (auto simp add:contains-def)
  have 3:s ⋆ (rs#r') = ⊥⋆(rs'#r') using 1 by fastforce
  show contains (s ⋆ (rs#r')) r using 2 3
    by (auto simp add:contains-def) (metis exec-cons rev-subsetD set-subset-Cons)
qed

lemma preceq-star: s ⋆ (rs#r) ≼ s' ⟹ s ⋆ rs ≼ s'
by (metis pre-RDR.exec.simps(1) pre-RDR.exec.simps(2) pre-RDR.less-eq-def trans)

end
```

4.3 The RDR locale

```
locale RDR = pre-RDR +
  assumes idem1:contains s r ⟹ s • r = s
  and idem2:⋀ s r r' . fst r ≠ fst r' ⟹ γ s r = γ ((s • r) • r') r
  and antisym:⋀ s1 s2 . s1 ≼ s2 ∧ s2 ≼ s1 ⟹ s1 = s2
  and glb-exists:⋀ s1 s2 . ∃ s . is-glb s s1 s2
  and consistency:⋀s1 s2 s3 rs . s1 ≼ s2 ⟹ s2 ≼ s3 ⟹ s3 = s1 ⋆ rs
    ⟹ ∃ rs' rs'' . s2 = s1 ⋆ rs' ∧ s3 = s2 ⋆ rs''
      ∧ set rs' ⊆ set rs ∧ set rs'' ⊆ set rs
  and bot:⋀ s . ⊥ ≼ s
begin

lemma inf-glb:is-glb (s1 ⊓ s2) s1 s2
```

```
proof −
  { fix s s′
    assume is-glb s s1 s2 and is-glb s′ s1 s2
    hence s = s′ using antisym by (auto simp add:is-glb-def is-lb-def) }
    from this and glb-exists show ?thesis
      by (auto simp add:inf-def, metis (lifting) theI′)
qed

sublocale ordering less-eq less
proof
  fix s
  show s ⪯ s
  by (metis exec.simps(1) less-eq-def)
next
  fix s s′
  show s ≺ s′ = (s ⪯ s′ ∧ s ≠ s′)
  by (auto simp add:less-def)
next
  fix s s′
  assume s ⪯ s′ and s′ ⪯ s
  thus s = s′
  using antisym by auto
next
  fix s1 s2 s3
  assume s1 ⪯ s2 and s2 ⪯ s3
  thus s1 ⪯ s3
  using trans by blast
qed

sublocale semilattice-set inf
proof
  fix s
  show s ⊓ s = s
    using inf-glb
    by (metis antisym is-glb-def is-lb-def refl)
next
  fix s1 s2
  show s1 ⊓ s2 = (s2 ⊓ s1)
    using inf-glb
    by (smt antisym is-glb-def pre-RDR.is-lb-def)
next
  fix s1 s2 s3
  show (s1 ⊓ s2) ⊓ s3 = (s1 ⊓ (s2 ⊓ s3))
    using inf-glb
    by(auto simp add:is-glb-def is-lb-def, smt antisym trans)
qed

sublocale semilattice-order-set inf less-eq less
proof
```

```
  fix s s′
  show s ⪯ s′ = (s = s ⊓ s′)
  by (metis antisym idem inf-glb pre-RDR.is-glb-def pre-RDR.is-lb-def)
next
  fix s s′
  show s ≺ s′ = (s = s ⊓ s′ ∧ s ≠ s′)
   by (metis inf-glb local.antisym local.refl pre-RDR.is-glb-def pre-RDR.is-lb-def
pre-RDR.less-def)
qed

notation F (‹⨅ -› [99])
```

4.4 Some useful lemmas

```
lemma idem-star:
fixes r s rs
assumes contains s r
shows s ⋆ rs = s ⋆ (filter (λ x . x ≠ r) rs)
proof (induct rs)
  case Nil
  show s ⋆ [] = s ⋆ (filter (λ x . x ≠ r) [])
    using assms by auto
next
  case (Cons r′ rs)
  have 1:contains (s ⋆ rs) r using assms and contains-star by auto
  show s ⋆ (rs#r′) = s ⋆ (filter (λ x . x ≠ r) (rs#r′))
  proof (cases r′ = r)
    case True
    hence s ⋆ (rs#r′) = s ⋆ rs using idem1 1 by auto
    thus ?thesis using Cons by simp
  next
    case False
    thus ?thesis using Cons by auto
  qed
qed

lemma idem-star2:
  fixes s rs′
  shows ∃ rs′ . s ⋆ rs = s ⋆ rs′ ∧ set rs′ ⊆ set rs
    ∧ (∀ r ∈ set rs′ . ¬ contains s r)
proof (induct rs)
  case Nil
  thus ∃ rs′ . s ⋆ [] = s ⋆ rs′ ∧ set rs′ ⊆ set []
    ∧ (∀ r ∈ set rs′ . ¬ contains s r) by force
next
  case (Cons r rs)
  obtain rs′ where 1:s ⋆ rs = s ⋆ rs′ and 2:set rs′ ⊆ set rs
  and 3:∀ r ∈ set rs′ . ¬ contains s r using Cons(1) by blast
  show ∃ rs′ . s ⋆ (rs#r) = s ⋆ rs′ ∧ set rs′ ⊆ set (rs#r)
```

```
  ∧ (∀ r ∈ set rs' . ¬ contains s r)
  proof (cases contains s r)
    case True
    have s⋆(rs#r) = s⋆rs'
    proof −
      have s ⋆ (rs#r) = s⋆rs using True
        by (metis contains-star exec-cons idem1)
      moreover
      have s ⋆ (rs'#r) = s⋆rs' using True
        by (metis contains-star exec-cons idem1)
      ultimately show ?thesis using 1 by simp
    qed
    moreover have set rs' ⊆ set (rs#r) using 2
      by (simp, metis subset-insertI2)
    moreover have ∀ r ∈ set rs' . ¬ contains s r
      using 3 by assumption
    ultimately show ?thesis by blast
  next
    case False
    have s⋆(rs#r) = s⋆(rs'#r) using 1 by simp
    moreover
    have set (rs'#r) ⊆ set (rs#r) using 2 by auto
    moreover have ∀ r ∈ set (rs'#r) . ¬ contains s r
      using 3 False by auto
    ultimately show ?thesis by blast
  qed
qed

lemma idem2-star:
assumes contains s r
and ⋀ r' . r' ∈ set rs ⟹ fst r' ≠ fst r
shows γ s r = γ (s ⋆ rs) r using assms
proof (induct rs)
  case Nil
  show γ s r = γ (s ⋆ []) r by simp
next
  case (Cons r' rs)
  thus γ s r = γ (s ⋆ (rs#r')) r
    using assms by auto
      (metis contains-star fst-conv idem1 idem2 prod.exhaust)
qed

lemma glb-common:
fixes s1 s2 s rs1 rs2
assumes s1 = s ⋆ rs1 and s2 = s ⋆ rs2
shows ∃ rs . s1 ⊓ s2 = s ⋆ rs ∧ set rs ⊆ set rs1 ∪ set rs2
proof −
  have 1:s ⪯ s1 and 2:s ⪯ s2 using assms by (auto simp add:less-eq-def)
  hence 3:s ⪯ s1 ⊓ s2 by (metis inf-glb is-lb-def pre-RDR.is-glb-def)
```

```
  have 4:s1 ⊓ s2 ≼ s1 by (metis cobounded1)
  show ?thesis using 3 4 assms(1) and consistency by blast
qed

lemma glb-common-set:
fixes ss s0 rset
assumes finite ss  and ss ≠ {}
and ⋀ s . s ∈ ss ⟹ ∃ rs . s = s0 ⋆ rs ∧ set rs ⊆ rset
shows ∃ rs . ⨅ ss = s0 ⋆ rs ∧ set rs ⊆ rset
using assms
proof (induct ss rule:finite-ne-induct)
  case (singleton s)
  obtain rs where s = s0 ⋆ rs ∧ set rs ⊆ rset using singleton by force
  moreover have ⨅ {s} = s using singleton by auto
  ultimately show ∃ rs . ⨅ {s} = s0 ⋆ rs ∧ set rs ⊆ rset by blast
next
  case (insert s ss)
  have 1:⋀ s' . s' ∈ ss ⟹ ∃ rs . s' = s0 ⋆ rs ∧ set rs ⊆ rset
    using insert(5) by force
  obtain rs where 2:⨅ ss = s0 ⋆ rs and 3:set rs ⊆ rset
    using insert(4) 1 by blast
  obtain rs' where 4:s = s0 ⋆ rs'and 5:set rs' ⊆ rset
    using insert(5) by blast
  have 6:⨅ (insert s ss) = s ⊓ (⨅ ss)
    by (metis insert.hyps(1−3) insert-not-elem)
  obtain rs'' where 7:⨅ (insert s ss) = s0 ⋆ rs''
    and 8:set rs'' ⊆ set rs' ∪ set rs
    using glb-common 2 4 6 by force
  have 9:set rs'' ⊆ rset using 3 5 8 by blast
  show ∃ rs . ⨅ (insert s ss) = s0 ⋆ rs ∧ set rs ⊆ rset
    using 7 9 by blast
qed

end

end
```

5 The SLin Automata specification

```
theory SLin
imports IOA RDR
begin

datatype ('a,'b,'c,'d)SLin-action =
— The nat component is the instance number
  Invoke nat 'b 'c
| Response nat 'b 'd
| Switch nat 'b 'c 'a
| Recover nat
```

```
| Linearize nat

datatype SLin-status = Sleep | Pending | Ready | Aborted

record ('a,'b,'c)SLin-state =
  pending :: 'b ⇒ 'b × 'c
  initVals :: 'a set
  abortVals :: 'a set
  status :: 'b ⇒ SLin-status
  dstate :: 'a
  initialized :: bool

locale SLin = RDR + IOA
begin

definition
  asig :: nat ⇒ nat ⇒ ('a,'b,'c,'d)SLin-action signature
  — The first instance has number 0
  where
  asig i j ≡ (|
    inputs = {act . ∃ p c iv i' .
      (i ≤ i' ∧ i' < j ∧ act = Invoke i' p c) ∨ (i > 0 ∧ act = Switch i p c iv)},
    outputs = {act . ∃ p c av i' outp .
      (i ≤ i' ∧ i' < j ∧ act = Response i' p outp) ∨ act = Switch j p c av},
    internals = {act. ∃ i' . i ≤ i' ∧ i' < j
      ∧ (act = Linearize i' ∨ act = Recover i')} |)

definition pendingReqs :: ('a,'b,'c)SLin-state ⇒ ('b×'c) set
  where
  pendingReqs s ≡ {r . ∃ p .
     r = pending s p
     ∧ status s p ∈ {Pending, Aborted}}

definition Inv :: 'b ⇒ 'c
  ⇒ ('a,'b,'c)SLin-state ⇒ ('a,'b,'c)SLin-state ⇒ bool
where
  Inv p c s s' ≡
    status s p = Ready
    ∧ s' = s(|pending := (pending s)(p := (p,c)),
        status := (status s)(p := Pending)|)

definition pendingSeqs where
  pendingSeqs s ≡ {rs . set rs ⊆ pendingReqs s}

definition Lin :: ('a,'b,'c)SLin-state ⇒ ('a,'b,'c)SLin-state ⇒ bool
where
  Lin s s' ≡ ∃ rs ∈ pendingSeqs s .
      initialized s
      ∧ (∀ av ∈ abortVals s . (dstate s) ⋆ rs ≼ av)
```

$\wedge\ s' = s(\!|dstate := (dstate\ s) \star rs|\!)$

definition *initSets* **where**
$initSets\ s \equiv \{ivs\ .\ ivs \neq \{\} \wedge ivs \subseteq initVals\ s\}$

definition *safeInits* **where**
$safeInits\ s \equiv if\ initVals\ s = \{\}\ then\ \{\}$
$else\ \{d\ .\ \exists\ ivs \in initSets\ s\ .\ \exists\ rs \in pendingSeqs\ s\ .$
$d = \bigsqcap ivs \star rs \wedge (\forall\ av \in abortVals\ s\ .\ d \preceq av)\}$

definition *initAborts* **where**
$initAborts\ s \equiv \{\ d\ .dstate\ s \preceq d$
$\wedge ((\exists\ rs \in pendingSeqs\ s\ .\ d = dstate\ s \star rs)$
$\vee (\exists\ ivs \in initSets\ s\ .\ dstate\ s \preceq \bigsqcap ivs$
$\wedge (\exists\ rs \in pendingSeqs\ s\ .\ d = \bigsqcap ivs \star rs)))\ \}$
definition *uninitAborts* **where**
$uninitAborts\ s \equiv \{\ d\ .$
$\exists\ ivs \in initSets\ s\ .\ \exists\ rs \in pendingSeqs\ s\ .$
$d = \bigsqcap ivs \star rs\ \}$

definition $safeAborts{::}('a,'b,'c)SLin\text{-}state \Rightarrow 'a\ set$ **where**
$safeAborts\ s \equiv if\ initialized\ s\ then\ initAborts\ s$
$else\ uninitAborts\ s$

definition $Reco :: ('a,'b,'c)SLin\text{-}state \Rightarrow ('a,'b,'c)SLin\text{-}state \Rightarrow bool$
where
$Reco\ s\ s' \equiv$
$(\exists\ p\ .\ status\ s\ p \neq Sleep)$
$\wedge \neg\ initialized\ s$
$\wedge (\exists\ d \in safeInits\ s\ .$
$s' = s(\!|dstate := d,\ initialized := True|\!))$

definition $Resp :: 'b \Rightarrow 'd \Rightarrow ('a,'b,'c)SLin\text{-}state \Rightarrow ('a,'b,'c)SLin\text{-}state \Rightarrow bool$
where
$Resp\ p\ ou\ s\ s' \equiv$
$status\ s\ p = Pending$
$\wedge\ initialized\ s$
$\wedge\ contains\ (dstate\ s)\ (pending\ s\ p)$
$\wedge\ ou = \gamma\ (dstate\ s)\ (pending\ s\ p)$
$\wedge\ s' = s\ (\!|status := (status\ s)(p := Ready)|\!)$

definition $Init :: 'b \Rightarrow 'c \Rightarrow 'a$
$\Rightarrow ('a,'b,'c)SLin\text{-}state \Rightarrow ('a,'b,'c)SLin\text{-}state \Rightarrow bool$
where
$Init\ p\ c\ iv\ s\ s' \equiv$
$status\ s\ p = Sleep$
$\wedge\ s' = s\ (\!|initVals := \{iv\} \cup (initVals\ s),$
$status := (status\ s)(p := Pending),$
$pending := (pending\ s)(p := (p,c))|\!)$

```
definition Abort :: 'b ⇒ 'c ⇒ 'a
  ⇒ ('a,'b,'c)SLin-state ⇒ ('a,'b,'c)SLin-state ⇒ bool
where
  Abort p c av s s' ≡
     status s p = Pending ∧ pending s p = (p,c)
     ∧ av ∈ safeAborts s
     ∧ s' = s(|status := (status s)(p := Aborted),
       abortVals := (abortVals s ∪ {av})|)

definition trans where
trans i j ≡ { (s,a,s') . case a of
  Invoke i' p c ⇒ i ≤ i' ∧ i < j ∧ Inv p c s s'
| Response i' p ou ⇒ i ≤ i' ∧ i < j ∧ Resp p ou s s'
| Switch i' p c v ⇒ (i > 0 ∧ i' = i ∧ Init p c v s s')
    ∨ (i' = j ∧ Abort p c v s s')
| Linearize i' ⇒ i' = i ∧ Lin s s'
| Recover i' ⇒ i > 0 ∧ i' = i ∧ Reco s s' }

definition start where
  start i ≡ { s .
    ∀ p . status s p = (if i > 0 then Sleep else Ready)
    ∧ dstate s = ⊥
    ∧ (if i > 0 then ¬ initialized s else initialized s)
    ∧ initVals s = {}
    ∧ abortVals s = {}}

definition ioa where
  ioa i j ≡
     (|ioa.asig = asig i j ,
       start = start i,
       trans = trans i j|)

end

end
```

6 Definition and Soundness of Refinement Mappings, Forward Simulations and Backward Simulations

```
theory Simulations
imports IOA
begin

context IOA
begin
```

```
definition refines where
  refines e s a t A f ≡ fst e = f s ∧ last-state e = f t ∧ is-exec-frag-of A e
          ∧ (let tr = trace (ioa.asig A) e in
              if a ∈ ext A then tr = [a] else tr = [])
definition
  is-ref-map :: ('s1 ⇒ 's2) ⇒ ('s1,'a)ioa ⇒ ('s2,'a)ioa ⇒ bool where
  is-ref-map f B A ≡
    (∀ s ∈ start B . f s ∈ start A) ∧ (∀ s t a. reachable B s ∧ s −a−B⟶ t
      ⟶ (∃ e . refines e s a t A f ))

definition
  is-forward-sim :: ('s1 ⇒ ('s2 set)) ⇒ ('s1,'a)ioa ⇒ ('s2,'a)ioa ⇒ bool where
  is-forward-sim f B A ≡
   (∀ s ∈ start B . f s ∩ start A ≠ {})
   ∧ (∀ s s' t a. s' ∈ f s ∧ s −a−B⟶ t ∧ reachable B s
       ⟶ (∃ e . fst e = s' ∧ last-state e ∈ f t ∧ is-exec-frag-of A e
            ∧ (let tr = trace (ioa.asig A) e in
                 if a ∈ ext A then tr = [a] else tr = [])))

definition
  is-backward-sim :: ('s1 ⇒ ('s2 set)) ⇒ ('s1,'a)ioa ⇒ ('s2,'a)ioa ⇒ bool where
  is-backward-sim f B A ≡
   (∀ s . f s ≠ {}) — Quantifying over reachable states would suffice
   ∧ (∀ s ∈ start B . f s ⊆ start A)
   ∧ (∀ s t a t'. t' ∈ f t ∧ s −a−B⟶ t ∧ reachable B s
       ⟶ (∃ e . fst e ∈ f s ∧ last-state e = t' ∧ is-exec-frag-of A e
             ∧ (let tr = trace (ioa.asig A) e in
                  if a ∈ ext A then tr = [a] else tr = [])))
```

6.1 A series of lemmas that will be useful in the soundness proofs

```
lemma step-eq-traces:
  fixes e-B' A e e-A' a t
  defines e-A ≡ append-exec e-A' e and e-B ≡ cons-exec e-B' (a,t)
  and tr ≡ trace (ioa.asig A) e
  assumes 1:trace (ioa.asig A) e-A' = trace (ioa.asig A) e-B'
  and 2:if a ∈ ext A then tr = [a] else tr = []
  shows trace (ioa.asig A) e-A = trace (ioa.asig A) e-B
proof −
  have 3:trace (ioa.asig A) e-B =
       (if a ∈ ext A then (trace (ioa.asig A) e-B') # a else trace (ioa.asig A) e-B')
    using e-B-def by (simp add:trace-def schedule-def filter-act-def cons-exec-def)
  have 4:trace (ioa.asig A) e-A =
        (if a ∈ ext A then trace (ioa.asig A) e-A' # a else trace (ioa.asig A) e-A')
    using 2 trace-append-is-append-trace[of ioa.asig A e-A' e]
      by(auto simp add:e-A-def tr-def split: if-split-asm)
  show ?thesis using 1 3 4 by simp
qed
```

```
lemma exec-inc-imp-trace-inc:
  fixes A B
  assumes ext B = ext A
  and ⋀ e-B . is-exec-of B e-B
    ⟹ ∃ e-A . is-exec-of A e-A ∧ trace (ioa.asig A) e-A = trace (ioa.asig A) e-B
  shows traces B ⊆ traces A
proof −
  { fix t
    assume t ∈ traces B
    with this obtain e where 1:t = trace (ioa.asig B) e and 2:is-exec-of B e
      using traces-alt assms(1) by blast
    from 1 and assms(1) have 3:t = trace (ioa.asig A) e by (simp add:trace-def)
    from 2 3 and assms(2) obtain e′ where
      is-exec-of A e′ ∧ trace (ioa.asig A) e′ = trace (ioa.asig A) e by blast
    hence t ∈ traces A using 3 traces-alt by fastforce }
  thus ?thesis by fast
qed
```

6.2 Soundness of Refinement Mappings

```
lemma ref-map-execs:
  fixes A::(′sA,′a)ioa and B::(′sB,′a)ioa and f::′sB ⇒ ′sA and e-B
  assumes is-ref-map f B A and is-exec-of B e-B
  shows ∃ e-A . is-exec-of A e-A
    ∧ trace (ioa.asig A) e-A = trace (ioa.asig A) e-B
proof −
  note assms(2)
  hence ∃ e-A . is-exec-of A e-A
                 ∧ trace (ioa.asig A) e-A = trace (ioa.asig A) e-B
                 ∧ last-state e-A = f (last-state e-B)
  proof (induction snd e-B arbitrary:e-B)
    case Nil
    let ?e-A = (f (fst e-B), [])
   have ⋀ s . s ∈ start B ⟹ f s ∈ start A using assms(1) by (simp add:is-ref-map-def)
    hence is-exec-of A ?e-A using Nil.prems(1) by (simp add:is-exec-of-def)
    moreover
    have trace (ioa.asig A) ?e-A = trace (ioa.asig A) e-B
      using Nil.hyps by (simp add:trace-simps)
    moreover
    have last-state ?e-A = f (last-state e-B)
      using Nil.hyps by (metis last-state.simps(1) prod.collapse)
    ultimately show ?case by fast
  next
    case (Cons p ps e-B)
    let ?e-B′ = (fst e-B, ps)
    let ?s = last-state ?e-B′ let ?t = snd p let ?a = fst p
    have 1:is-exec-of B ?e-B′ and 2:?s−?a−B⟶?t
      using Cons.prems and Cons.hyps(2)
```

```
        by (simp-all add:is-exec-of-def,
            cases (B,fst e-B,ps#p) rule:is-exec-frag-of.cases, auto,
            cases (B,fst e-B,ps#p) rule:is-exec-frag-of.cases, auto)
    with Cons.hyps(1) obtain e-A' where ih1:is-exec-of A e-A'
      and ih2:trace (ioa.asig A) e-A' = trace (ioa.asig A) ?e-B'
      and ih3:last-state e-A' = f ?s by fastforce
    from 1 have 3:reachable B ?s using last-state-reachable by fast
    obtain e where 4:fst e = f ?s and 5:last-state e = f ?t
    and 6:is-exec-frag-of A e
    and 7:let tr = trace (ioa.asig A) e in if ?a ∈ ext A
      then tr = [?a] else tr = []
        using 2 and 3 and assms(1)
          by (force simp add:is-ref-map-def refines-def)
    let ?e-A = append-exec e-A' e
    have is-exec-of A ?e-A
      using ih1 ih3 4 6 append-exec-frags-is-exec-frag[of A e e-A']
        by (metis append-exec-def append-exec-frags-is-exec-frag
          fst-conv is-exec-of-def)
    moreover
    have trace (ioa.asig A) ?e-A = trace (ioa.asig A) e-B
      using ih2 Cons.hyps(2) 7 step-eq-traces[of A e-A' ?e-B' ?a e]
        by (auto simp add:cons-exec-def) (metis prod.collapse)
    moreover have last-state ?e-A = f ?t using ih3 4 5 last-state-of-append
      by metis
    ultimately show ?case using Cons.hyps(2)
      by (metis last-state.simps(2) surjective-pairing)
  qed
  thus ?thesis by blast
qed

theorem ref-map-soundness:
  fixes A::('sA,'a)ioa and B::('sB,'a)ioa and f::'sB ⇒ 'sA
  assumes is-ref-map f B A and ext A = ext B
  shows traces B ⊆ traces A
  using assms ref-map-execs exec-inc-imp-trace-inc by metis
```

6.3 Soundness of Forward Simulations

```
lemma forward-sim-execs:
  fixes A::('sA,'a)ioa and B::('sB,'a)ioa and f::'sB ⇒ 'sA set and e-B
  assumes is-forward-sim f B A and is-exec-of B e-B
  shows ∃ e-A . is-exec-of A e-A
    ∧ trace (ioa.asig A) e-A = trace (ioa.asig A) e-B
proof –
  note assms(2)
  hence ∃ e-A . is-exec-of A e-A
                ∧ trace (ioa.asig A) e-A = trace (ioa.asig A) e-B
                ∧ last-state e-A ∈ f (last-state e-B)
  proof (induction snd e-B arbitrary:e-B)
```

```
    case Nil
    have ⋀ s . s ∈ start B ⟹ f s ∩ start A ≠ {}
      using assms(1) by (simp add:is-forward-sim-def)
    with this obtain s′ where 1:s′ ∈ f (fst e-B) and 2:s′ ∈ start A
      by (metis Int-iff Nil.prems all-not-in-conv is-exec-of-def)
    let ?e-A = (s′, [])
    have is-exec-of A ?e-A using 2 by (simp add:is-exec-of-def)
    moreover
    have trace (ioa.asig A) ?e-A = trace (ioa.asig A) e-B using Nil.hyps
      by (simp add:trace-def schedule-def filter-act-def)
    moreover
    have last-state ?e-A ∈ f (last-state e-B)
      using Nil.hyps 1 by (metis last-state.simps(1) surjective-pairing)
    ultimately show ?case by fast
  next
    case (Cons p ps e-B)
    let ?e-B′ = (fst e-B, ps)
    let ?s = last-state ?e-B′ let ?t = snd p let ?a = fst p
    have 1:is-exec-of B ?e-B′ and 2:?s−?a−B⟶?t
      using Cons.prems and Cons.hyps(2)
        by (simp-all add:is-exec-of-def,
            cases (B,fst e-B,ps#p) rule:is-exec-frag-of.cases, auto,
            cases (B,fst e-B,ps#p) rule:is-exec-frag-of.cases, auto)
    with Cons.hyps(1) obtain e-A′ where ih1:is-exec-of A e-A′
      and ih2:trace (ioa.asig A) e-A′ = trace (ioa.asig A) ?e-B′
      and ih3:last-state e-A′ ∈ f ?s by fastforce
    from 1 have 3:reachable B ?s using last-state-reachable by fast
    obtain e where 4:fst e = last-state e-A′ and 5:last-state e ∈ f ?t
    and 6:is-exec-frag-of A e
    and 7:let tr = trace (ioa.asig A) e in if ?a ∈ ext A then tr = [?a] else tr = []
      using 2 3 assms(1) ih3 by (simp add:is-forward-sim-def)
        (metis prod.collapse prod.inject)
    let ?e-A = append-exec e-A′ e
    have is-exec-of A ?e-A
      using ih1 ih3 4 6 append-exec-frags-is-exec-frag[of A e e-A′]
        by (metis append-exec-def append-exec-frags-is-exec-frag
            fst-conv is-exec-of-def)
    moreover
    have trace (ioa.asig A) ?e-A = trace (ioa.asig A) e-B
      using ih2 Cons.hyps(2) 7 step-eq-traces[of A e-A′ ?e-B′ ?a e]
        by (auto simp add:cons-exec-def Let-def) (metis prod.collapse)
    moreover have last-state ?e-A ∈ f ?t using ih3 4 5 last-state-of-append
      by metis
    ultimately show ?case using Cons.hyps(2)
      by (metis last-state.simps(2) surjective-pairing)
  qed
  thus ?thesis by blast
qed
```

```
theorem forward-sim-soundness:
  fixes A::('sA,'a)ioa and B::('sB,'a)ioa and f::'sB ⇒ 'sA set
  assumes is-forward-sim f B A and ext A = ext B
  shows traces B ⊆ traces A
  using assms forward-sim-execs exec-inc-imp-trace-inc by metis
```

6.4 Soundness of Backward Simulations

```
lemma backward-sim-execs:
  fixes A::('sA,'a)ioa and B::('sB,'a)ioa and f::'sB ⇒ 'sA set and e-B
  assumes is-backward-sim f B A and is-exec-of B e-B
  shows ∃ e-A . is-exec-of A e-A
    ∧ trace (ioa.asig A) e-A = trace (ioa.asig A) e-B
proof −
  note assms(2)
  hence ∀ s ∈ f (last-state e-B). ∃ e-A .
        is-exec-of A e-A
        ∧ trace (ioa.asig A) e-A = trace (ioa.asig A) e-B
        ∧ last-state e-A = s
  proof (induction snd e-B arbitrary:e-B)
    case Nil
    { fix s' assume 1:s' ∈ f(last-state e-B)
      have 2:⋀ s . s ∈ start B ⟹ f s ⊆ start A
        using assms(1) by (simp add:is-backward-sim-def)
      from Nil 1 2 have 3:s' ∈ start A
           by (metis (full-types) is-exec-of-def last-state.simps(1) subsetD surjective-pairing)
      let ?e-A = (s', [])
      have 4:is-exec-of A ?e-A using 3 by (simp add:is-exec-of-def)
      have 5:trace (ioa.asig A) ?e-A = trace (ioa.asig A) e-B using Nil.hyps
        by (simp add:trace-def schedule-def filter-act-def)
      have 6:last-state ?e-A ∈ f (last-state e-B)
        using Nil.hyps 1 by (metis last-state.simps(1))
      note 4 5 6 }
    thus ?case by fastforce
  next
    case (Cons p ps e-B)
    { fix t' assume 8:t' ∈ f (last-state e-B)
      let ?e-B' = (fst e-B, ps)
      let ?s = last-state ?e-B' let ?t = snd p let ?a = fst p
      have 5:?t = last-state e-B using Cons.hyps(2)
        by (metis last-state.simps(2) prod.collapse)
      have 1:is-exec-of B ?e-B' and 2:?s−?a−B⟶?t
        using Cons.prems and Cons.hyps(2)
          by (simp-all add:is-exec-of-def,
              cases (B,fst e-B, ps#p) rule:is-exec-frag-of.cases, auto,
              cases (B,fst e-B, ps#p) rule:is-exec-frag-of.cases, auto)
      from 1 have 3:reachable B ?s using last-state-reachable by fast
      obtain e where 4:fst e ∈ f ?s and 5:last-state e = t'
```

```
      and 6:is-exec-frag-of A e
      and 7:let tr = trace (ioa.asig A) e in
        if ?a ∈ ext A then tr = [?a] else tr = []
        using 2 assms(1) 8 5 3 by (auto simp add: is-backward-sim-def, metis)
      obtain e-A′ where ih1:is-exec-of A e-A′
        and ih2:trace (ioa.asig A) e-A′ = trace (ioa.asig A) ?e-B′
        and ih3:last-state e-A′ = fst e
          using 1 4 Cons.hyps(1) by (metis snd-conv)
      let ?e-A = append-exec e-A′ e
      have is-exec-of A ?e-A
        using ih1 ih3 4 6 append-exec-frags-is-exec-frag[of A e e-A′]
          by (metis append-exec-def append-exec-frags-is-exec-frag
            fst-conv is-exec-of-def)
      moreover
      have trace (ioa.asig A) ?e-A = trace (ioa.asig A) e-B
        using ih2 Cons.hyps(2) 7 step-eq-traces[of A e-A′ ?e-B′ ?a e]
          by (auto simp add:cons-exec-def Let-def) (metis prod.collapse)
      moreover have last-state ?e-A = t′ using ih3 5 last-state-of-append
        by metis
      ultimately have ∃ e-A . is-exec-of A e-A
        ∧ trace (ioa.asig A) e-A = trace (ioa.asig A) e-B
        ∧ last-state e-A = t′ by blast }
    thus ?case by blast
  qed
  moreover
  from assms(1) have total:⋀ s . f s ≠ {} by (simp add:is-backward-sim-def)
  ultimately show ?thesis by fast
qed

theorem backward-sim-soundness:
  fixes A::('sA,'a)ioa and B::('sB,'a)ioa and f::'sB ⇒ 'sA set
  assumes is-backward-sim f B A and ext A = ext B
  shows traces B ⊆ traces A
  using assms backward-sim-execs exec-inc-imp-trace-inc by metis

end

end
```

7 Idempotence of the SLin I/O automaton

```
theory Idempotence
imports SLin Simulations
begin

locale Idempotence = SLin +
  fixes id1 id2 :: nat
  assumes id1:0 < id1 and id2:id1 < id2
begin
```

lemmas *ids* = *id1 id2*

definition *composition* **where**
composition $\equiv$
hide ((*ioa 0 id1*) $\|$ (*ioa id1 id2*))
$\{act\ .\ \exists\, p\ c\ av\ .\ act = Switch\ id1\ p\ c\ av\ \}$

lemmas *comp-simps* = *hide-def composition-def ioa-def par2-def is-trans-def*
start-def actions-def asig-def trans-def

lemmas *trans-defs* = *Inv-def Lin-def Resp-def Init-def*
Abort-def Reco-def

declare *if-split-asm* [*split*]

7.1 A case rule for decomposing the transition relation of the composition of two SLins

declare *comp-simps* [*simp*]
lemma *trans-elim*:
fixes $s\ t\ a\ s'\ t'\ P$
assumes $(s,t)\ {-a-}composition{\longrightarrow}\ (s',t')$
obtains
(*Invoke1*) $i\ p\ c$
where $Inv\ p\ c\ s\ s' \wedge t = t'$
and $i < id1$ **and** $a = Invoke\ i\ p\ c$
| (*Invoke2*) $i\ p\ c$
where $Inv\ p\ c\ t\ t' \wedge s = s'$
and $id1 \leq i \wedge i < id2$ **and** $a = Invoke\ i\ p\ c$
| (*Switch1*) $p\ c\ av$
where $Abort\ p\ c\ av\ s\ s' \wedge Init\ p\ c\ av\ t\ t'$
and $a = Switch\ id1\ p\ c\ av$
| (*Switch2*) $p\ c\ av$
where $s = s' \wedge Abort\ p\ c\ av\ t\ t'$
and $a = Switch\ id2\ p\ c\ av$
| (*Response1*) $i\ p\ ou$
where $Resp\ p\ ou\ s\ s' \wedge t = t'$
and $i < id1$ **and** $a = Response\ i\ p\ ou$
| (*Response2*) $i\ p\ ou$
where $Resp\ p\ ou\ t\ t' \wedge s = s'$
and $id1 \leq i \wedge i < id2$ **and** $a = Response\ i\ p\ ou$
| (*Lin1*) $Lin\ s\ s' \wedge t = t'$ **and** $a = Linearize\ 0$
| (*Lin2*) $Lin\ t\ t' \wedge s = s'$ **and** $a = Linearize\ id1$
| (*Reco2*) $Reco\ t\ t' \wedge s = s'$ **and** $a = Recover\ id1$
declare *comp-simps* [*simp del*]

7.2 Definition of the Refinement Mapping

```
fun f :: (('a,'b,'c)SLin-state ∗ ('a,'b,'c)SLin-state) ⇒ ('a,'b,'c)SLin-state
  where
  f (s1, s2) =
     (|pending = λ p. (if status s1 p ≠ Aborted then pending s1 p else pending s2 p),
      initVals = {},
      abortVals = abortVals s2,
      status = λ p. (if status s1 p ≠ Aborted then status s1 p else status s2 p),
      dstate = (if dstate s2 = ⊥ then dstate s1 else dstate s2),
      initialized = True|)
```

7.3 Invariants

```
declare
  trans-defs [simp]

fun P1 where
  P1 (s1,s2) = (∀ p . status s1 p ∈ {Pending, Aborted}
    ⟶ fst (pending s1 p) = p)

fun P2 where
  P2 (s1,s2) = (∀ p . status s2 p ≠ Sleep ⟶ fst (pending s2 p) = p)

fun P3 where
  P3 (s1,s2) = (∀ p . (status s2 p = Ready ⟶ initialized s2))


fun P4 where
  P4 (s1,s2) = ((∀ p . status s2 p = Sleep) = (initVals s2 = {}))

fun P5 where
  P5 (s1,s2) = (∀ p . status s1 p ≠ Sleep ∧ initialized s1 ∧ initVals s1 = {})


fun P6  where
  P6 (s1,s2) = (∀ p . (status s1 p ≠ Aborted) = (status s2 p = Sleep))

fun P7 where
  P7 (s1,s2) = (∀ c . status s1 c = Aborted ∧ ¬ initialized s2
    ⟶ (pending s2 c = pending s1 c ∧ status s2 c ∈ {Pending, Aborted}))


fun P8 where
  P8 (s1,s2) = (∀ iv ∈ initVals s2 . ∃ rs ∈ pendingSeqs s1 .
    iv = dstate s1 ⋆ rs)

fun P8a where
  P8a (s1,s2) = (∀ ivs ∈ initSets s2 . ∃ rs ∈ pendingSeqs s1 .
    ⨅ ivs = dstate s1 ⋆ rs)
```

fun *P9* **where**
$P9\ (s1,s2) = (initialized\ s2 \longrightarrow dstate\ s1 \preceq dstate\ s2)$

fun *P10* **where**
$P10\ (s1,s2) = ((\neg\ initialized\ s2) \longrightarrow (dstate\ s2 = \bot))$

fun *P11* **where**
$P11\ (s1,s2) = (initVals\ s2 = abortVals\ s1)$

fun *P12* **where**
$P12\ (s1,s2) = (initialized\ s2 \longrightarrow \bigsqcap\ (initVals\ s2) \preceq dstate\ s2)$

fun *P13* **where**
$P13\ (s1,s2) = (finite\ (initVals\ s2)$
$\wedge\ finite\ (abortVals\ s1) \wedge finite\ (abortVals\ s2))$

fun *P14* **where**
$P14\ (s1,s2) = (initialized\ s2 \longrightarrow initVals\ s2 \neq \{\})$

fun *P15* **where**
$P15\ (s1,s2) = (\forall\ av \in abortVals\ s1\ .\ dstate\ s1 \preceq av)$

fun *P16* **where**
$P16\ (s1,s2) = (dstate\ s2 \neq \bot \longrightarrow initialized\ s2)$

fun *P17* **where**
— For the Response1 case of the refinement proof, in case a response is produced in the first instance and the second instance is already initialized

$$\begin{aligned}
&P17\ (s1,s2) = (initialized\ s2\\
&\quad \longrightarrow (\forall\ p\ .\\
&\qquad ((status\ s1\ p = Ready\\
&\qquad\quad \vee\ (status\ s1\ p = Pending \wedge contains\ (dstate\ s1)\ (pending\ s1\ p)))\\
&\qquad\qquad \longrightarrow (\exists\ rs\ .\ dstate\ s2 = dstate\ s1 \star rs \wedge (\forall\ r \in set\ rs\ .\ fst\ r \neq p)))\\
&\qquad \wedge\ ((status\ s1\ p = Pending \wedge \neg\ contains\ (dstate\ s1)\ (pending\ s1\ p))\\
&\qquad\quad \longrightarrow (\exists\ rs\ .\ dstate\ s2 = dstate\ s1 \star rs \wedge (\forall\ r \in set\ rs\ .\\
&\qquad\qquad fst\ r = p \longrightarrow r = pending\ s1\ p)))))
\end{aligned}$$

fun *P18* **where**
$P18\ (s1,s2) = (abortVals\ s2 \neq \{\} \longrightarrow (\exists\ p\ .\ status\ s2\ p \neq Sleep))$

fun *P19* **where**
$P19\ (s1,s2) = (abortVals\ s2 \neq \{\} \longrightarrow abortVals\ s1 \neq \{\})$

fun *P20* **where**
$P20\ (s1,s2) = (\forall\ av \in abortVals\ s2\ .\ dstate\ s2 \preceq av)$

fun *P21* **where**

$P21\ (s1,s2) = (\forall\ av \in abortVals\ s2\ .\ \bigsqcap(abortVals\ s1) \preceq av)$

fun *P22* **where**
$P22\ (s1,s2) = (initialized\ s2 \longrightarrow dstate\ (f\ (s1,s2)) = dstate\ s2)$

fun *P23* **where**
$P23\ (s1,s2) = ((\neg\ initialized\ s2) \longrightarrow$
$pendingSeqs\ s1 \subseteq pendingSeqs\ (f\ (s1,s2)))$

fun *P25* **where**
$P25\ (s1,s2) = (\forall\ ivs\ .\ (ivs \in initSets\ s2 \wedge initialized\ s2$
$\wedge\ dstate\ s2 \preceq \bigsqcap ivs)$
$\longrightarrow (\exists\ rs' \in pendingSeqs\ (f\ (s1,s2))\ .\ \bigsqcap ivs = dstate\ s2 \star rs'))$

fun *P26* **where**
$P26\ (s1,s2) = (\forall\ p\ .\ (status\ s1\ p = Aborted$
$\wedge\ \neg\ contains\ (dstate\ s2)\ (pending\ s1\ p))$
$\longrightarrow (status\ s2\ p \in \{Pending,Aborted\}$
$\wedge\ pending\ s1\ p = pending\ s2\ p))$

lemma *P1-invariant*:
shows *invariant* (*composition*) *P1*
proof (*rule invariantI*, *simp-all only*:*split-paired-all*)
fix *s1 s2*
assume $(s1,s2) \in ioa.start\ (composition)$
thus $P1\ (s1,s2)$ **using** *ids* **by** (*auto simp add*:*comp-simps*)
next
fix *s1 s2 t1 t2 a*
assume *hyp*: $P1\ (s1,s2)$ **and** *trans*:$(s1,s2)\ {-a-}composition{\longrightarrow}\ (t1,t2)$
show $P1\ (t1,t2)$ **using** *trans* **and** *hyp*
by (*cases rule*:*trans-elim*) *auto*
qed

lemma *P2-invariant*:
shows *invariant* (*composition*) *P2*
proof (*rule invariantI*, *simp-all only*:*split-paired-all*)
fix *s1 s2*
assume $(s1,s2) \in ioa.start\ (composition)$
thus $P2\ (s1,s2)$ **using** *ids* **by** (*auto simp add*:*comp-simps*)
next
fix *s1 s2 t1 t2 a*
assume *hyp*: $P2\ (s1,s2)$ **and** *trans*:$(s1,s2)\ {-a-}composition{\longrightarrow}\ (t1,t2)$
show $P2\ (t1,t2)$ **using** *trans* **and** *hyp*
by (*cases rule*:*trans-elim*) *auto*
qed

lemma *P16-invariant*:
shows *invariant* (*composition*) *P16*
proof (*rule invariantI*, *simp-all only*:*split-paired-all*)

```
  fix s1 s2
  assume (s1,s2) ∈ ioa.start (composition)
  thus P16 (s1,s2) using ids by (auto simp add:comp-simps)
next
  fix s1 s2 t1 t2 a
  assume hyp: P16 (s1,s2) and trans:(s1,s2) −a−composition−→ (t1,t2)
  show P16 (t1,t2) using trans and hyp
  by (cases rule:trans-elim) auto
qed

lemma P3-invariant:
shows invariant (composition) P3
proof (rule invariantI, simp-all only:split-paired-all)
  fix s1 s2
  assume (s1,s2) ∈ ioa.start (composition)
  thus P3 (s1,s2) using ids by (auto simp add:comp-simps)
next
  fix s1 s2 t1 t2 a
  assume hyp: P3 (s1,s2) and trans:(s1,s2) −a−composition−→ (t1,t2)
  show P3 (t1,t2) using trans and hyp
  by (cases rule:trans-elim) auto
qed

lemma P4-invariant:
shows invariant (composition) P4
proof (rule invariantI, simp-all only:split-paired-all)
  fix s1 s2
  assume (s1,s2) ∈ ioa.start (composition)
  thus P4 (s1,s2) using ids by (auto simp add:comp-simps)
next
  fix s1 s2 t1 t2 a
  assume hyp: P4 (s1,s2) and trans:(s1,s2) −a−composition−→ (t1,t2)
  show P4 (t1,t2) using trans and hyp
  by (cases rule:trans-elim) auto
qed

lemma P5-invariant:
shows invariant (composition) P5
proof (rule invariantI, simp-all only:split-paired-all)
  fix s1 s2
  assume (s1,s2) ∈ ioa.start (composition)
  thus P5 (s1,s2) using ids by (auto simp add:comp-simps)
next
  fix s1 s2 t1 t2 a
  assume hyp: P5 (s1,s2) and trans:(s1,s2) −a−composition−→ (t1,t2)
  show P5 (t1,t2) using trans and hyp
  by (cases rule:trans-elim) auto
qed
```

```
lemma P13-invariant:
shows invariant (composition) P13
proof (rule invariantI, simp-all only:split-paired-all)
  fix s1 s2
  assume (s1,s2) ∈ ioa.start (composition)
  thus P13 (s1,s2) using ids by (auto simp add:comp-simps)
next
  fix s1 s2 t1 t2 a
  assume hyp: P13 (s1,s2) and trans:(s1,s2) −a−composition−→ (t1,t2)
  show P13 (t1,t2) using trans and hyp
  by (cases rule:trans-elim, auto)
qed

lemma P20-invariant:
shows invariant (composition) P20
proof (rule invariantI, simp-all only:split-paired-all)
  fix s1 s2
  assume (s1,s2) ∈ ioa.start (composition)
  thus P20 (s1,s2) using ids by (auto simp add:comp-simps)
next
  fix s1 s2 t1 t2 a
  assume hyp: P20 (s1,s2) and trans:(s1,s2) −a−composition−→ (t1,t2)
  and reach: reachable (composition) (s1,s2)
  from reach have P16:P16 (s1,s2) using P16-invariant and ids
    by (metis IOA.invariant-def)
  show P20 (t1,t2) using trans and hyp and P16
  by (cases rule:trans-elim, auto simp add:safeInits-def safeAborts-def
    initAborts-def uninitAborts-def bot)
qed

lemma P18-invariant:
shows invariant (composition) P18
proof (rule invariantI, simp-all only:split-paired-all)
  fix s1 s2
  assume (s1,s2) ∈ ioa.start (composition)
  thus P18 (s1,s2) using ids by (auto simp add:comp-simps)
next
  fix s1 s2 t1 t2 a
  assume hyp: P18 (s1,s2) and trans:(s1,s2) −a−composition−→ (t1,t2)
  show P18 (t1,t2) using trans and hyp
  by (cases rule:trans-elim, auto)
qed

lemma P14-invariant:
shows invariant (composition) P14
proof (rule invariantI, simp-all only:split-paired-all)
  fix s1 s2
  assume (s1,s2) ∈ ioa.start (composition)
```

```
  thus P14 (s1,s2) using ids by (auto simp add:comp-simps)
next
  fix s1 s2 t1 t2 a
  assume hyp: P14 (s1,s2) and trans:(s1,s2) −a−composition−→ (t1,t2)
  show P14 (t1,t2) using trans and hyp
  by (cases rule:trans-elim, auto simp add:safeInits-def)
qed

lemma P15-invariant:
shows invariant (composition) P15
proof (rule invariantI, simp-all only:split-paired-all)
  fix s1 s2
  assume (s1,s2) ∈ ioa.start (composition)
  thus P15 (s1,s2) using ids by (auto simp add:comp-simps)
next
  fix s1 s2 t1 t2 a
  assume hyp: P15 (s1,s2) and trans:(s1,s2) −a−composition−→ (t1,t2)
  and reach: reachable (composition) (s1,s2)
  from reach have P5:P5 (s1,s2) using P5-invariant and ids
    by (metis IOA.invariant-def)
  show P15 (t1,t2) using trans and hyp and P5
  by (cases rule:trans-elim,
    auto simp add:less-eq-def safeAborts-def initAborts-def)
qed

lemma P6-invariant:
shows invariant (composition) P6
proof (rule invariantI, simp-all only:split-paired-all)
  fix s1 s2
  assume (s1,s2) ∈ ioa.start (composition)
  thus P6 (s1,s2) using ids by (auto simp add:comp-simps)
next
  fix s1 s2 t1 t2 a
  assume hyp: P6 (s1,s2) and trans:(s1,s2) −a−composition−→ (t1,t2)
  show P6 (t1,t2) using trans and hyp
  by (cases rule:trans-elim, force+)
qed

lemma P7-invariant:
shows invariant (composition) P7
proof (rule invariantI, simp-all only:split-paired-all)
  fix s1 s2
  assume (s1,s2) ∈ ioa.start (composition)
  thus P7 (s1,s2) using ids by (auto simp add:comp-simps)
next
  fix s1 s2 t1 t2 a
  assume hyp: P7 (s1,s2) and trans:(s1,s2) −a−composition−→ (t1,t2)
  show P7 (t1,t2) using trans and hyp
  by (cases rule:trans-elim) auto
```

```
qed

lemma P10-invariant:
shows invariant (composition) P10
proof (rule invariantI, simp-all only:split-paired-all)
  fix s1 s2
  assume (s1,s2) ∈ ioa.start (composition)
  thus P10 (s1,s2) using ids by (auto simp add:comp-simps)
next
  fix s1 s2 t1 t2 a
  assume hyp: P10 (s1,s2) and trans:(s1,s2) −a−composition⟶ (t1,t2)
  show P10 (t1,t2) using trans and hyp
  by (cases rule:trans-elim) auto
qed

lemma P11-invariant:
shows invariant (composition) P11
proof (rule invariantI, simp-all only:split-paired-all)
  fix s1 s2
  assume (s1,s2) ∈ ioa.start (composition)
  thus P11 (s1,s2) using ids by (auto simp add:comp-simps)
next
  fix s1 s2 t1 t2 a
  assume hyp: P11 (s1,s2) and trans:(s1,s2) −a−composition⟶ (t1,t2)
  show P11 (t1,t2) using trans and hyp
  by (cases rule:trans-elim, force+)
qed

lemma P8-invariant:
shows invariant (composition) P8
proof (rule invariantI, simp-all only:split-paired-all)
  fix s1 s2
  assume (s1,s2) ∈ ioa.start (composition)
  thus P8 (s1,s2) using ids by (auto simp add:comp-simps)
next
  fix s1 s2 t1 t2 a
  assume hyp: P8 (s1,s2) and trans:(s1,s2) −a−composition⟶ (t1,t2)
  and reach: reachable (composition) (s1,s2)
  from reach have P5:P5 (s1,s2) using P5-invariant and ids
    by (metis IOA.invariant-def)
  from reach have P1:P1 (s1,s2) using P1-invariant and ids
    by (metis IOA.invariant-def)
  from reach have P11:P11 (s1,s2) using P11-invariant and ids
    by (metis IOA.invariant-def)
  show P8 (t1,t2) using trans and hyp
  proof (cases rule:trans-elim)
    case (Invoke1 i p c)
    assume P8 (s1,s2)
    have pendingSeqs s1 ⊆ pendingSeqs t1
```

```
    proof −
      have pending t1 = (pending s1)(p := (p,c))
      and status t1 = (status s1)(p := Pending)
      and status s1 p = Ready
      using Invoke1(1) by auto
     hence pendingReqs s1 ⊆ pendingReqs t1 by (simp add:pendingReqs-def) force
      thus ?thesis by (auto simp add:pendingSeqs-def)
    qed
    moreover have initVals t2 = initVals s2 and dstate t1 = dstate s1
      using Invoke1(1) by auto
    ultimately show P8 (t1,t2) using ‹P8 (s1,s2)› by fastforce
  next
    case Lin1
    assume P8 (s1,s2)
    show P8 (t1,t2)
    proof (simp, rule ballI)
      fix iv
      assume 0:iv ∈ initVals t2
      have 1:iv ∈ initVals s2 using Lin1(1) 0 by simp
      have 4:iv ∈ abortVals s1 using 1 P11 by simp
      obtain rs where 2:rs ∈ pendingSeqs s1 and 3:iv = dstate s1 ⋆ rs
        using ‹P8 (s1,s2)› 1 by auto
      obtain rs' where 6:dstate t1 = dstate s1 ⋆ rs' and 5:dstate s1 ⋆ rs' ≼ iv
        using Lin1(1) 1 4 by auto
      obtain rs'' where 7:iv = (dstate s1 ⋆ rs') ⋆ rs'' and 8:set rs'' ⊆ set rs
        using consistency 3 5 6 by simp (metis less-eq-def)
      have 10:rs'' ∈ pendingSeqs t1
      proof −
        have 9:pendingSeqs t1 = pendingSeqs s1
          using Lin1(1) by (auto simp add:pendingSeqs-def pendingReqs-def)
        thus ?thesis using 8 2 by (auto simp add:pendingSeqs-def)
      qed
      show ∃ rs ∈ pendingSeqs t1 . iv = dstate t1 ⋆ rs
        using 7 10 6 by auto
    qed
  next
    case (Response1 i p ou)
    assume ih:P8 (s1,s2)
    show P8 (t1,t2)
    proof auto
      fix iv
      assume 1:iv ∈ initVals t2
      obtain rs where 2:iv = dstate t1 ⋆ rs and 3:rs ∈ pendingSeqs s1
        using 1 Response1(1) ih by auto
      have 4:pendingReqs t1 = ((pendingReqs s1) − {pending s1 p})
      proof −
        have pending t1 = pending s1 and status t1 = (status s1)(p := Ready)
          and 5:status s1 p = Pending
          using Response1(1) by auto
```

```
      moreover have ⋀ q . q ≠ p ⟹ status s1 q ∈ {Pending,Aborted}
        ⟹ pending s1 q ≠ pending s1 p
        using P1 5 by (metis P1.simps insertI1)
      ultimately show ?thesis by (simp add:pendingReqs-def) fastforce
    qed
    have 8:contains (dstate t1) (pending s1 p) using Response1(1) by simp
    define rs′ where rs′ = filter (λ x . x ≠ (pending s1 p)) rs
    have 9:rs′ ∈ pendingSeqs t1
    proof –
      have 9:pending s1 p ∉ set rs′ by (auto simp add:rs′-def)
      have 10:rs′ ∈ pendingSeqs s1
        using 3 by (auto simp add:rs′-def)
          (metis filter-is-subset mem-Collect-eq pendingSeqs-def subset-trans)
      show ?thesis using 10 9 4 by (auto simp add:pendingSeqs-def)
    qed
    have 10:iv = dstate t1 ⋆ rs′ using 8 2 idem-star rs′-def by fast
    show ∃ rs ∈ pendingSeqs t1 . iv = dstate t1 ⋆ rs using 10 9 by auto
  qed
next
  case (Switch1 p c av)
  assume P8 (s1,s2)
  have 1:initialized s1 ∧ initVals s1 = {} using P5 by auto
  obtain av where 2:initVals t2 = initVals s2 ∪ {av} and 3:av ∈ safeAborts s1
    using Switch1(1) by auto
  obtain rs where 4:rs ∈ pendingSeqs s1 and 5:av = dstate s1 ⋆ rs
    using 1 3 by (auto simp add:safeAborts-def initAborts-def initSets-def)
  have 6:dstate s1 = dstate t1 using Switch1(1) by simp
  have 7:pendingSeqs t1 = pendingSeqs s1
  proof –
    have pendingReqs t1 = pendingReqs s1
      using Switch1(1) by (simp add:pendingReqs-def) fastforce
    thus ?thesis by (auto simp add:pendingSeqs-def)
  qed
  show P8 (t1,t2) using ‹P8 (s1,s2)› 2 4 5 6 7 by auto
next
  case (Invoke2 i p c)
  assume P8 (s1,s2)
  thus P8 (t1,t2) using Invoke2(1) by force
next
  case Lin2
  assume P8 (s1,s2)
  thus P8 (t1,t2) using Lin2(1) by force
next
  case (Response2 i p ou)
  assume P8 (s1,s2)
  thus P8 (t1,t2) using Response2(1) by force
next
  case (Switch2 p c av)
  assume P8 (s1,s2)
```

```
    thus P8 (t1,t2) using Switch2(1) by force
  next
    case Reco2
    assume P8 (s1,s2)
    thus P8 (t1,t2) using Reco2(1) by force
  qed
qed

lemma P8a-invariant:
shows invariant (composition) P8a
proof (auto simp:invariant-def)
  fix s1 s2 ivs
  assume 1:reachable (composition) (s1,s2)
  and 2:ivs ∈ initSets s2
  have 3:finite ivs ∧ ivs ≠ {}
  proof −
    have P13 (s1,s2) using P13-invariant 1
      by (metis IOA.invariant-def)
    thus ?thesis using 2 finite-subset by (auto simp add:initSets-def)
  qed
  have 4:∀ av ∈ ivs . ∃ rs ∈ pendingSeqs s1 . av = dstate s1 ⋆ rs
  proof −
    have P8:P8 (s1,s2) using P8-invariant 1
      by (metis IOA.invariant-def)
    thus ?thesis using 2 by (auto simp add:initSets-def)
  qed
  show ∃ rs ∈ pendingSeqs s1 . ⨅ ivs = dstate s1 ⋆ rs
    using 3 4 glb-common-set by (simp add:pendingSeqs-def, metis)
qed

lemma P12-invariant:
shows invariant (composition) P12
proof (rule invariantI, simp-all only:split-paired-all)
  fix s1 s2
  assume (s1,s2) ∈ ioa.start (composition)
  thus P12 (s1,s2) using ids by (auto simp add:comp-simps)
next
  fix s1 s2 t1 t2 a
  assume hyp: P12 (s1,s2) and trans:(s1,s2) −a−composition⟶ (t1,t2)
  and reach: reachable (composition) (s1,s2)
  from reach have P13:P13 (s1,s2) using P13-invariant
    by (metis IOA.invariant-def)
  from reach have P14:P14 (s1,s2) using P14-invariant
    by (metis IOA.invariant-def)
  show P12 (t1,t2) using trans and hyp
  proof (cases rule:trans-elim)
    case (Invoke1 i p c)
    assume P12 (s1,s2)
    thus P12 (t1,t2) using Invoke1(1) by auto
```

```
  next
    case Lin1
    assume P12 (s1,s2)
    thus P12 (t1,t2) using Lin1(1) by auto
  next
    case (Response1 i p ou)
    assume P12 (s1,s2)
    thus P12 (t1,t2) using Response1(1) by auto
  next
    case (Switch1 p c av)
    assume ih:P12 (s1,s2)
    have initialized s2 ⟹ ⨅ (initVals t2) ≼ ⨅ (initVals s2)
    proof -
      assume 1:initialized s2
      have initVals t2 = initVals s2 ∪ {av} using Switch1(1) by simp
      hence ⨅ (initVals t2) = ⨅ (initVals s2) ⊓ av
        using insert-not-elem P13 P14 1
         by (metis P13.simps P14.simps Un-empty-right Un-insert-right commute
insert)
      thus ?thesis by (metis cobounded1)
    qed
    moreover have dstate t2 = dstate s2 and initialized s2 = initialized t2
      using Switch1(1) by auto
    ultimately show P12 (t1,t2) using ih by auto (metis absorb2 coboundedI1)
  next
    case (Invoke2 i p c)
    assume P12 (s1,s2)
    thus P12 (t1,t2) using Invoke2(1) by force
  next
    case Lin2
    assume P12 (s1,s2)
    moreover
    have initVals t2 = initVals s2 and initialized s2
    and initialized t2 using Lin2(1) by auto
    moreover
    have dstate s2 ≼ dstate t2 using Lin2(1) by auto (metis less-eq-def)
    ultimately show P12 (t1,t2) by auto (metis strict-iff-order strict-trans1)
  next
    case (Response2 i p ou)
    assume P12 (s1,s2)
    thus P12 (t1,t2) using Response2(1) by force
  next
    case (Switch2 p c av)
    assume P12 (s1,s2)
    thus P12 (t1,t2) using Switch2(1) by force
  next
    case Reco2
    obtain d where 1:d ∈ safeInits s2 and 2:dstate t2 = d
      using Reco2(1) by force
```

```
    obtain ivs where 3:ivs ⊆ initVals s2 and 4:ivs ≠ {}
      and 5:⨅ivs ≼ d
      using 1 by (auto simp add:safeInits-def initSets-def)
        (metis equals0D less-eq-def)
    have 6:⨅ (initVals s2) ≼ ⨅ ivs using 3 P13 4
      by (simp add: subset-imp)
    have 7:initVals s2 = initVals t2 using Reco2(1) by auto
    show P12 (t1,t2) using 2 5 6 7
      by (metis P12.simps absorb2 coboundedI1)
  qed
qed

lemma P19-invariant:
shows invariant (composition) P19
proof (auto simp only:invariant-def)
  fix s1 s2
  assume 1:reachable (composition) (s1,s2)
  have P4:P4 (s1,s2) using P4-invariant 1
    by (simp add:invariant-def)
  moreover
  have P18:P18 (s1,s2) using P18-invariant 1
    by (metis IOA.invariant-def)
  moreover
  have P11:P11 (s1,s2) using P11-invariant 1
    by (metis IOA.invariant-def)
  moreover
  ultimately show P19 (s1,s2) by auto
qed

lemma P9-invariant:
shows invariant (composition) P9
proof (auto simp only:invariant-def)
  fix s1 s2
  assume 1:reachable (composition) (s1,s2)
  have P12:P12 (s1,s2) using P12-invariant 1
    by (simp add:invariant-def)
  have P15:P15 (s1,s2) using P15-invariant 1
    by (metis IOA.invariant-def)
  have P13:P13 (s1,s2) using P13-invariant 1
    by (metis IOA.invariant-def)
  have P14:P14 (s1,s2) using P14-invariant 1
    by (metis IOA.invariant-def)
  have P11:P11 (s1,s2) using P11-invariant 1
    by (metis IOA.invariant-def)
  have initialized s2 ⟹ dstate s1 ≼ ⨅(abortVals s1)
    using P13 P15 P14 P11 boundedI by simp
  thus P9 (s1,s2) using P12 P11 by simp (metis trans)
qed
```

```
lemma P17-invariant:
shows invariant (composition) P17
proof (rule invariantI, simp-all only:split-paired-all)
  fix s1 s2
  assume (s1,s2) ∈ ioa.start (composition)
  thus P17 (s1,s2) using ids by (auto simp add:comp-simps)
next
  fix s1 s2 t1 t2 a
  assume hyp: P17 (s1,s2) and trans:(s1,s2) −a−composition⟶ (t1,t2)
  and reach:reachable (composition) (s1,s2)
  show P17 (t1,t2) using trans and hyp
  proof (cases rule:trans-elim)
    case (Invoke1 i p c)
    assume P17 (s1,s2)
    thus P17 (t1,t2) using Invoke1(1) by fastforce
  next
    case (Response1 i p ou)
    assume P17 (s1,s2)
    thus P17 (t1,t2) using Response1(1) by auto
  next
    case (Switch1 p c av)
    assume P17 (s1,s2)
    thus P17 (t1,t2) using Switch1(1) by auto
  next
    case (Invoke2 i p c)
    assume P17 (s1,s2)
    thus P17 (t1,t2) using Invoke2(1) by force
  next
    case (Response2 i p ou)
    assume P17 (s1,s2)
    thus P17 (t1,t2) using Response2(1) by force
  next
    case (Switch2 p c av)
    assume P17 (s1,s2)
    thus P17 (t1,t2) using Switch2(1) by force
  next
    case Lin1
    assume 1:P17 (s1,s2)
    obtain rs' where 2:dstate t1 = dstate s1 ⋆ rs'
      using Lin1(1) 1 by auto
    have 3:dstate s2 = dstate t2 using Lin1(1) by auto
    have 4:initialized t2 ⟹ dstate t1 ⪯ dstate t2
    proof −
      assume initialized t2
      moreover
      have P9 (t1,t2) using reach trans P9-invariant
        by (metis IOA.invariant-def reachable-n)
      ultimately show ?thesis by auto
    qed
```

```
show P17 (t1,t2)
proof(simp, auto)
  fix p
  assume 5:initialized t2 and 6:status t1 p = Ready
  obtain rs where 7:∀ r ∈ set rs . fst r ≠ p
    and 8:dstate t2 = dstate s1 ⋆ rs
  proof −
    obtain rs where dstate s2 = dstate s1 ⋆ rs
      ∧ (∀ r ∈ set rs . fst r ≠ p) using 1 5 6 Lin1(1) by force
    hence ∀ r ∈ set rs . fst r ≠ p and dstate t2 = dstate s1 ⋆ rs
      using Lin1(1) by auto
    thus ?thesis using that by blast
  qed
  have 9:dstate t1 ≼ dstate t2 using 4 5 by auto
  obtain rs'' where 10:dstate t2 = dstate t1 ⋆ rs''
    and 11:set rs'' ⊆ set rs
      using consistency 2 8 9 by simp (metis less-eq-def)
  have 12:∀ r ∈ set rs'' . fst r ≠ p using 7 11 by blast
  thus ∃ rs . dstate t2 = dstate t1 ⋆ rs ∧ (∀ r ∈ set rs . fst r ≠ p)
    using 10 12 by auto
next
  fix p
  assume 5:initialized t2 and 6:status t1 p = Pending
    and 7:¬ contains (dstate t1) (pending t1 p)
  obtain rs where 8:∀ r ∈ set rs . fst r = p ⟶ r = pending s1 p
    and 9:dstate t2 = dstate s1 ⋆ rs
  proof −
    have 9:¬ contains (dstate s1) (pending s1 p)
      using 7 Lin1(1) contains-star by fastforce
    obtain rs where dstate s2 = dstate s1 ⋆ rs
      ∧ (∀ r ∈ set rs . fst r = p ⟶ r = pending s1 p)
        using 1 5 6 9 Lin1(1) by force
    hence ∀ r ∈ set rs . fst r = p ⟶ r = pending s1 p
      and dstate t2 = dstate s1 ⋆ rs
        using Lin1(1) by auto
    thus ?thesis using that by blast
  qed
  have 10:dstate t1 ≼ dstate t2 using 4 5 by auto
  obtain rs'' where 11:dstate t2 = dstate t1 ⋆ rs''
    and 12:set rs'' ⊆ set rs
      using consistency 2 9 10 by simp (metis less-eq-def)
  have 13:∀ r ∈ set rs'' . fst r = p ⟶ r = pending s1 p
    using 8 12 by blast
  show ∃ rs . dstate t2 = dstate t1 ⋆ rs
    ∧ (∀ r ∈ set rs . fst r = p ⟶ r = pending t1 p)
    using 11 13 Lin1(1) by auto
next
  fix p
  assume 5:initialized t2 and 6:status t1 p = Pending
```

```
  and 7:contains (dstate t1) (pending t1 p)
show ∃ rs . dstate t2 = dstate t1 ⋆ rs
  ∧ (∀ r ∈ set rs . fst r ≠ p)
proof (cases contains (dstate s1) (pending s1 p))
  case True
  obtain rs where 8:∀ r ∈ set rs . fst r ≠ p
    and 9:dstate t2 = dstate s1 ⋆ rs
  proof −
    obtain rs where dstate s2 = dstate s1 ⋆ rs
      ∧ (∀ r ∈ set rs . fst r ≠ p) using 1 5 6 True Lin1(1) by force
    hence ∀ r ∈ set rs . fst r ≠ p and dstate t2 = dstate s1 ⋆ rs
      using Lin1(1) by auto
    thus ?thesis using that by blast
  qed
  have 10:dstate t1 ⪯ dstate t2 using 4 5 by auto
  obtain rs'' where 11:dstate t2 = dstate t1 ⋆ rs''
    and 12:set rs'' ⊆ set rs
      using consistency 2 9 10 by simp (metis less-eq-def)
  have 13:∀ r ∈ set rs'' . fst r ≠ p using 8 12 by blast
  thus ∃ rs . dstate t2 = dstate t1 ⋆ rs ∧ (∀ r ∈ set rs . fst r ≠ p)
    using 11 13 by auto
next
  case False
  obtain rs'' where 8:dstate t2 = dstate t1 ⋆ rs''
    and 9:∀ r ∈ set rs'' . fst r = p ⟶ r = pending t1 p
  proof −
    obtain rs where 8:∀ r ∈ set rs . fst r = p ⟶ r = pending s1 p
      and 9:dstate t2 = dstate s1 ⋆ rs
    proof −
      obtain rs where dstate s2 = dstate s1 ⋆ rs
        ∧ (∀ r ∈ set rs . fst r = p ⟶ r = pending s1 p)
          using 1 5 6 False Lin1(1) by force
      hence ∀ r ∈ set rs . fst r = p ⟶ r = pending s1 p
        and dstate t2 = dstate s1 ⋆ rs
          using Lin1(1) by auto
      thus ?thesis using that by blast
    qed
    have 10:dstate t1 ⪯ dstate t2 using 4 5 by auto
    obtain rs'' where 11:dstate t2 = dstate t1 ⋆ rs''
      and 12:set rs'' ⊆ set rs
        using consistency 2 9 10 by simp (metis less-eq-def)
    have 13:∀ r ∈ set rs'' . fst r = p ⟶ r = pending s1 p
      using 8 12 by blast
    have dstate t2 = dstate t1 ⋆ rs''
      ∧ (∀ r ∈ set rs'' . fst r = p ⟶ r = pending t1 p)
        using 11 13 Lin1(1) by auto
    thus ?thesis using that by blast
  qed
  have 10:dstate t1 ⋆ rs''
```

```
        = dstate t1 ⋆ (filter (λ r . r ≠ pending t1 p) rs'')
        using 7 idem-star by blast
      have 11:∀ r ∈ set (filter (λ r . r ≠ pending t1 p) rs'') .
        fst r ≠ p using 9 by force
      show ∃ rs . dstate t2 = dstate t1 ⋆ rs ∧ (∀ r ∈ set rs . fst r ≠ p)
        using 8 10 11 by metis
    qed
  qed
next
  case Lin2
  assume 1:P17 (s1,s2)
  { fix p
    assume 2:status s1 p ≠ Aborted
    have ∃ rs' . dstate t2 = dstate s2 ⋆ rs'
      ∧ (∀ r ∈ set rs' . fst r ≠ p)
    proof −
      obtain rs' where 5:dstate t2 = dstate s2 ⋆ rs'
        and 6:rs' ∈ pendingSeqs s2 using Lin2(1) by force
      have 7:∀ r ∈ set rs' . fst r ≠ p
      proof (rule ballI)
        fix r
        assume r ∈ set rs'
        with 6 have r ∈ pendingReqs s2 by (auto simp add:pendingSeqs-def)
        moreover
        have P2 (s1,s2) using reach P2-invariant
          by (metis invariant-def)
        moreover
        have status s2 p = Sleep
        proof −
          have P6 (s1,s2) using reach P6-invariant
            by (metis invariant-def)
          thus ?thesis using 2 Lin2(1) by force
        qed
        ultimately show fst r ≠ p by (auto simp add:pendingReqs-def)
      qed
      show ?thesis using 5 7 by force
    qed }
    note a = this
  show P17 (t1,t2)
  proof auto
    fix p
    assume 2:initialized t2 and 3:status t1 p = Ready
    obtain rs where dstate s2 = dstate s1 ⋆ rs
      and ∀ r ∈ set rs . fst r ≠ p
    proof −
      have initialized s2 and status s1 p = Ready
        using Lin2(1) 2 3 by auto
      thus ?thesis using that 1 by fastforce
    qed
```

```
    moreover
    obtain rs' where dstate t2 = dstate s2 ⋆ rs'
      and ∀ r ∈ set rs' . fst r ≠ p using a 3 Lin2(1)
        by (metis SLin-status.distinct(11))
    ultimately show ∃ rs . dstate t2 = dstate t1 ⋆ rs
      ∧ (∀ r ∈ set rs . fst r ≠ p) using Lin2(1)
        by auto (metis UnE exec-append set-append)
  next
    fix p
    assume 2:initialized t2 and 3:status t1 p = Pending
      and 4:contains (dstate t1) (pending t1 p)
    obtain rs where dstate s2 = dstate s1 ⋆ rs
      and ∀ r ∈ set rs . fst r ≠ p
    proof −
      have initialized s2 and status s1 p = Pending
        ∧ contains (dstate s1) (pending s1 p)
          using Lin2(1) 2 3 4 by auto
      thus ?thesis using that 1 by fastforce
    qed
    moreover
    obtain rs' where dstate t2 = dstate s2 ⋆ rs'
      and ∀ r ∈ set rs' . fst r ≠ p using a 3 Lin2(1)
        by (metis SLin-status.distinct(9))
    ultimately show ∃ rs . dstate t2 = dstate t1 ⋆ rs
      ∧ (∀ r ∈ set rs . fst r ≠ p) using Lin2(1)
        by auto (metis UnE exec-append set-append)
  next
    fix p
    assume 2:initialized t2 and 3:status t1 p = Pending
      and 4:¬ contains (dstate t1) (pending t1 p)
    obtain rs where dstate s2 = dstate s1 ⋆ rs
      and ∀ r ∈ set rs . fst r = p ⟶ r = pending s1 p
    proof −
      have initialized s2 and status s1 p = Pending
        ∧ ¬ contains (dstate s1) (pending s1 p)
          using Lin2(1) 2 3 4 by auto
      thus ?thesis using that 1 by fastforce
    qed
    moreover
    obtain rs' where dstate t2 = dstate s2 ⋆ rs'
      and ∀ r ∈ set rs' . fst r ≠ p using a 3 Lin2(1)
        by (metis SLin-status.distinct(9))
    ultimately show ∃ rs . dstate t2 = dstate t1 ⋆ rs
      ∧ (∀ r ∈ set rs . fst r = p ⟶ r = pending t1 p)
        using Lin2(1)
          by auto (metis UnE exec-append set-append)
  qed
next
  case Reco2
```

```
assume 0:P17 (s1,s2)
obtain rs′ where 1:dstate t2 = dstate t1 ⋆ rs′
  and 2:set rs′ ⊆ pendingReqs s1 ∪ pendingReqs s2
proof −
  obtain ivs rs where 1:ivs ⊆ initVals s2 and 2:ivs ≠ {}
    and 3:dstate t2 = ⨅ ivs ⋆ rs and 4:rs ∈ pendingSeqs s2
    using Reco2(1) by (simp add:safeInits-def initSets-def, force)
  obtain rs′′ where set rs′′ ⊆ pendingReqs s1
    and ⨅ ivs = dstate s1 ⋆ rs′′
  proof −
    have P8a (s1,s2) using reach P8a-invariant
      by (metis invariant-def)
    thus ?thesis using that using 1 2
      by (auto simp add:initSets-def pendingSeqs-def)
  qed
  hence dstate t2 = dstate t1 ⋆ (rs′′@rs)
    ∧ set rs′′ ⊆ pendingReqs s1
      ∧ set rs ⊆ pendingReqs s2
    using 3 4 Reco2(1) 4
      by (metis exec-append mem-Collect-eq pendingSeqs-def)
  thus ?thesis using that by force
qed
{ fix p r
  assume 1:r ∈ pendingReqs s2
    and 2:status s1 p ≠ Aborted
  have fst r ≠ p
  proof −
    have P2 (s1,s2) using reach P2-invariant
      by (metis invariant-def)
    moreover
    have P6 (s1,s2) using reach P6-invariant
      by (metis invariant-def)
    ultimately show ?thesis using 1 2 Reco2(1)
      by (simp add:pendingReqs-def)
        (metis SLin-status.distinct(1,5))
    qed }
note 3 = this
{ fix r p
  assume 1:r ∈ pendingReqs s1 and 2:fst r = p
    and 3:status s1 p = Pending
  have r = pending s1 p
  proof −
    have P1 (s1,s2) using reach P1-invariant
      by (metis invariant-def)
    thus ?thesis using 1 2 3
      by (auto simp add:pendingReqs-def)
  qed }
note 10 = this
show P17 (t1,t2)
```

```
proof (auto)
 fix p
 assume 4:status t1 p = Ready
 show ∃ rs . dstate t2 = dstate t1 ⋆ rs
   ∧ (∀ r ∈ set rs . fst r ≠ p)
 proof −
   { fix r
     assume 5:r ∈ pendingReqs s1
     have fst r ≠ p
     proof −
       have P1 (s1,s2) using reach P1-invariant
         by (metis invariant-def)
       with 4 5 Reco2(1) show ?thesis
         by (auto simp add:pendingReqs-def)
     qed }
   moreover
   have ⋀ r . r ∈ pendingReqs s2 ⟹ fst r ≠ p
     using 3 4 Reco2(1) by auto
   ultimately show ?thesis using 1 2 by blast
 qed
next
 fix p
 assume 4:status t1 p = Pending
   and 5:contains (dstate t1) (pending t1 p)
 show ∃ rs . dstate t2 = dstate t1 ⋆ rs
   ∧ (∀ r ∈ set rs . fst r ≠ p)
 proof −
   let ?rs = filter (λ r . r ≠ pending t1 p) rs'
   have dstate t2 = dstate t1 ⋆ ?rs
     using 5 1 idem-star by metis
   moreover
   { fix r
     assume r ∈ set ?rs
     have fst r ≠ p
     proof −
       { fix r
         assume 6:r ∈ set rs' and 7:fst r = p
         have r = pending s1 p
         proof −
           have ⋀ r . r ∈ pendingReqs s2 ⟹ fst r ≠ p
             using 3 4 Reco2(1) by auto
           moreover
           have ⋀ r . r ∈ pendingReqs s1 ⟹ fst r = p
             ⟹ r = pending s1 p
               using 10 4 Reco2(1) by auto
           ultimately show ?thesis using 2 6 7
             by (metis (lifting, no-types) UnE subsetD)
         qed }
       thus ?thesis using ‹r ∈ set ?rs› Reco2(1) by fastforce
```

```
          qed }
        ultimately show ?thesis by blast
      qed
    next
      fix p
      assume 4:status t1 p = Pending
        and 5:¬ contains (dstate t1) (pending t1 p)
      show ∃ rs . dstate t2 = dstate t1 ⋆ rs
        ∧ (∀ r ∈ set rs . fst r = p ⟶ r = pending t1 p)
      proof −
        have ⋀ r . r ∈ pendingReqs s2 ⟹ fst r ≠ p
          using 3 4 Reco2(1) by auto
        moreover
        have ⋀ r . r ∈ pendingReqs s1 ⟹ fst r = p
          ⟹ r = pending s1 p
            using 10 4 Reco2(1) by auto
        ultimately show ?thesis using 1 2 Reco2(1)
          by (metis (lifting, no-types) UnE rev-subsetD)
      qed
    qed
  qed
qed

lemma P21-invariant:
shows invariant (composition) P21
proof (rule invariantI, simp-all only:split-paired-all)
  fix s1 s2
  assume (s1,s2) ∈ ioa.start (composition)
  thus P21 (s1,s2) using ids by (auto simp add:comp-simps)
next
  fix s1 s2 t1 t2 a
  assume hyp: P21 (s1,s2) and trans:(s1,s2) −a−composition⟶ (t1,t2)
  and reach: reachable (composition) (s1,s2)
  show P21 (t1,t2)
  proof (cases initialized t2)
    case True
    moreover
    have P12:P12 (t1,t2) using P12-invariant reach trans
      by (metis invariant-def reachable-n)
    moreover
    have P11:P11 (t1,t2) using P11-invariant reach trans
      by (metis IOA.invariant-def reachable-n)
    moreover
    have P20:P20 (t1,t2) using P20-invariant reach trans
      by (metis IOA.invariant-def reachable-n)
    ultimately show P21 (t1,t2) by simp
      (metis pre-RDR.trans)
  next
    case False
```

```
show P21 (t1,t2) using trans
proof (cases rule:trans-elim)
  case (Switch2 p c av)
  obtain av where abortVals t2 = abortVals s2 ∪ {av}
    and ⨅(abortVals s1) ⪯ av
  proof −
    obtain ivs rs where 1:abortVals t2 = abortVals s2 ∪ {⨅ivs ⋆ rs}
      and 2:ivs ⊆ initVals s2 and 3:ivs ≠ {}
      using False Switch2(1) by (auto simp add:safeAborts-def
        uninitAborts-def initSets-def)
    have 4:⨅(abortVals s1) ⪯ ⨅ivs
    proof −
      have P11 (s1,s2) using reach P11-invariant
        by (metis invariant-def)
      moreover
      have P13 (s1,s2) using reach P13-invariant
        by (metis invariant-def)
      ultimately show ?thesis
        using 2 3 by (simp add: subset-imp)
    qed
    show ?thesis using that 1 4 by simp
      (metis coboundedI2 less-eq-def orderE)
  qed
  with hyp show ?thesis using Switch2(1) by simp
next
  case (Switch1 p c av)
  show ?thesis
  proof (cases abortVals s1 = {})
    case False
    have ⨅ (abortVals t1) ⪯ ⨅ (abortVals s1)
    proof −
      obtain av where abortVals t1 = abortVals s1 ∪ {av}
        using Switch1(1) by auto
      moreover
      have P13 (s1,s2) using reach P13-invariant
        by (metis invariant-def)
      ultimately show ?thesis using False by simp
    qed
    moreover have abortVals t2 = abortVals s2
      using Switch1(1) by auto
    ultimately show ?thesis using hyp
      by auto (metis coboundedI2 orderE)
  next
    case True
    have abortVals t2 = {}
    proof −
      have P19 (s1,s2) using reach P19-invariant
        by (metis invariant-def)
      thus ?thesis using True Switch1(1) by auto
```

```
        qed
        thus ?thesis by auto
      qed
    next
      case (Invoke1 p c)
      thus ?thesis using hyp by simp
    next
      case (Invoke2 p c)
      thus ?thesis using hyp by simp
    next
      case (Response1 p ou)
      thus ?thesis using hyp by simp
    next
      case (Response2 p ou)
      thus ?thesis using hyp by simp
    next
      case Lin1
      thus ?thesis using hyp by auto
    next
      case Lin2
      thus ?thesis using hyp by auto
    next
      case Reco2
      thus ?thesis using hyp by auto
    qed
  qed
qed

lemma P22-invariant:
shows invariant (composition) P22
proof (auto simp only:invariant-def)
  fix s1 s2
  assume 1:reachable (composition) (s1,s2)
  have P9:P9 (s1,s2) using P9-invariant 1
    by (simp add:invariant-def)
  show P22 (s1,s2)
  proof (simp only:P22.simps, rule impI)
    assume initialized s2
    show dstate (f (s1,s2)) = dstate s2
    proof (cases dstate s2 = ⊥)
      case False
      thus ?thesis by auto
    next
      case True
      show dstate  (f (s1,s2)) = dstate s2
      proof −
        have dstate s1 ⪯ dstate s2
          using ‹initialized s2› and ‹P9 (s1,s2)›
            by auto
```

```
        hence dstate s1 = dstate s2 using True
          by (metis antisym bot)
        thus ?thesis by auto
      qed
    qed
  qed
qed

lemma P23-invariant:
shows invariant (composition) P23
proof (auto simp only:invariant-def)
  fix s1 s2
  assume 1:reachable (composition) (s1,s2)
  show P23 (s1,s2)
  proof (simp only:P23.simps, clarify)
    fix rs
    assume 2:¬initialized s2 and 3:rs∈pendingSeqs s1
    show rs∈ pendingSeqs (f (s1,s2))
    proof −
      { fix r
        assume 3:r ∈ pendingReqs s1
        have 4:status s1 (fst r) = Pending ∨ status s1 (fst r) = Aborted
          and 5:pending s1 (fst r) = r
        proof −
          have P1 (s1,s2) using 1 P1-invariant
            by (metis invariant-def)
          thus status s1 (fst r) = Pending ∨ status s1 (fst r) = Aborted
          and pending s1 (fst r) = r
            using 3 by (auto simp add:pendingReqs-def)
        qed
        have r ∈ pendingReqs (f (s1,s2)) using 4
        proof
          assume status s1 (fst r) = Pending
          with 5 show ?thesis by (auto simp add:pendingReqs-def)
            (metis SLin-status.distinct(9))
        next
          assume 6:status s1 (fst r) = Aborted
          have 7:pending s1 (fst r) = pending s2 (fst r)
            ∧ status s2 (fst r) ∈ {Pending,Aborted}
          proof −
            have P7 (s1,s2) using 1 P7-invariant
              by (metis invariant-def)
            thus ?thesis using 2 6 by auto
          qed
          show ?thesis using 6 5 7 by (simp add:pendingReqs-def, metis)
        qed }
      thus ?thesis using 3 by (auto simp only:pendingSeqs-def)
    qed
  qed
```

```
qed

lemma P26-invariant:
shows invariant (composition) P26
proof (rule invariantI, simp-all only:split-paired-all)
  fix s1 s2
  assume (s1,s2) ∈ ioa.start (composition)
  thus P26 (s1,s2) using ids by (auto simp add:comp-simps)
next
  fix s1 s2 t1 t2 a
  assume hyp: P26 (s1,s2) and trans:(s1,s2) −a−composition−→ (t1,t2)
  and reach:reachable composition (s1,s2)
  show P26 (t1,t2) using trans and hyp
  proof (cases rule:trans-elim)
    case Lin2
    hence 1:dstate s2 ⪯ dstate t2
      by auto (metis less-eq-def)
    have 2:t2 = s2⦇dstate := dstate t2⦈ and 3:s1 = t1
      using Lin2(1) by auto
    show ?thesis
    proof (simp, clarify)
      fix p
      assume 4:status t1 p = Aborted
        and 5:¬ contains (dstate t2) (pending t1 p)
      have 6:status s1 p = Aborted using 3 4 by auto
      have 7:pending s1 p = pending t1 p using 3 by simp
      have 8:¬ contains (dstate s2) (pending s1 p)
        using 1 5 7
          by simp (metis contains-star less-eq-def)
      have 11:status s2 p ∈ {Pending,Aborted}
        and 9:pending s1 p = pending s2 p using hyp 6 8 by auto
      show (status t2 p = Pending ∨ status t2 p = Aborted)
        ∧ pending t1 p = pending t2 p
      proof −
        from 2 have pending s2 = pending t2
            and status s2 = status t2 by ((cases s2, cases t2, auto)+)
        thus ?thesis using 9 3 11 by auto
      qed
    qed
  next
    case Reco2
    show ?thesis
    proof (simp,clarify)
      fix p
      assume 1:status t1 p = Aborted
      have 2:status s1 p = Aborted and 3:¬initialized s2
        using 1 Reco2(1) by auto
      have 4:P7 (s1,s2) using reach P7-invariant
        by (metis invariant-def)
```

```
      have 5:status s2 p ∈ {Pending,Aborted}
      and 6:pending s1 p = pending s2 p using 3 4 2 by auto
      show (status t2 p = Pending ∨ status t2 p = Aborted)
        ∧ pending t1 p = pending t2 p using 5 6 Reco2(1) by auto
    qed
  next
    case Lin1
    thus ?thesis using hyp by force
  next
    case Response1
    thus ?thesis using hyp by force
  next
    case Response2
    thus ?thesis using hyp by force
  next
    case Invoke2
    thus ?thesis using hyp by force
  next
    case Switch1
    thus ?thesis using hyp by force
  next
    case Switch2
    thus ?thesis using hyp by force
  next
    case Invoke1
    thus ?thesis using hyp by force
  qed
qed

lemma P25-invariant:
shows invariant (composition) P25
proof (auto simp only:invariant-def)
  fix s1 s2
  assume reach:reachable (composition) (s1,s2)
  show P25 (s1,s2)
  proof (simp only:P25.simps, clarify)
    fix ivs
    assume 1:ivs ∈ initSets s2 and 2:initialized s2
      and 3:dstate s2 ⪯ ⨅ivs
    obtain rs' where 4:dstate s2 ⋆ rs' = ⨅ivs
    and 5:rs' ∈ pendingSeqs s1 and 6:∀ r ∈ set rs' . ¬ contains (dstate s2) r
    proof −
      have 5:dstate s1 ⪯ dstate s2
      proof −
        have P9:P9 (s1,s2) using P9-invariant reach
          by (simp add:invariant-def)
        thus ?thesis using 2 by auto
      qed
      obtain rs where 6:⨅ivs = dstate s1 ⋆ rs and 7:rs ∈ pendingSeqs s1
```

```
      proof –
        have P8a:P8a (s1,s2) using P8a-invariant reach
          by (simp add:invariant-def)
        thus ?thesis using that 1 by auto
      qed
      have ∃ rs′ . dstate s2 ⋆ rs′ = ⨅ ivs ∧ rs′ ∈ pendingSeqs s1
        using 3 5 6 7 consistency[of dstate s1 dstate s2 ⨅ ivs rs]
          by (force simp add:pendingSeqs-def)
      with this obtain rs′ where ⨅ ivs = dstate s2 ⋆ rs′
        and rs′ ∈ pendingSeqs s1 by metis
      with this show ?thesis using idem-star2 that
        by (metis mem-Collect-eq pendingSeqs-def subset-trans)
    qed
    have 7:rs′ ∈ pendingSeqs (f (s1,s2))
    proof –
      { fix r
        assume r ∈ set rs′
        with this obtain p where 8:status s1 p = Pending
          ∨ status s1 p = Aborted
        and 9:r = pending s1 p
          using 5 by (auto simp add:pendingReqs-def pendingSeqs-def)
        from 8 have r ∈ pendingReqs (f (s1,s2))
        proof
          assume status s1 p = Pending
          thus ?thesis using 9 by (simp add:pendingReqs-def)
            (metis SLin-status.distinct(9))
        next
          assume 10:status s1 p = Aborted
          hence status (f (s1,s2)) p = status s2 p
            and pending (f (s1,s2)) p = pending s2 p by simp-all
          moreover
          have status s2 p ∈ {Pending,Aborted} ∧ pending s2 p = pending s1 p
          proof –
            have ¬ contains (dstate s2) r
              using 6 ‹r ∈ set rs′› by simp
            moreover
            have P26 (s1,s2) using reach P26-invariant
              by (metis invariant-def)
            ultimately show ?thesis using 10 9 by force
          qed
          ultimately show ?thesis using 9 by (simp only:pendingReqs-def, force)
        qed }
      thus ?thesis by (auto simp add:pendingSeqs-def)
    qed
    show ∃ rs ∈ pendingSeqs (f (s1,s2)) . ⨅ ivs = dstate s2 ⋆ rs
      using 4 7 by force
  qed
qed
```

7.4 Proof of the Idempotence Theorem

```
theorem idempotence:
  shows ((composition) =<| (ioa 0 id2))
proof −
  have same-input-sig:inp (composition) = inp (ioa 0 id2)
    — First we show that both automata have the same input and output signature
      using  ids by auto
  moreover
  have same-output-sig:out (composition) = out (ioa 0 id2)
    — Then we show that output signatures match
    using ids by auto
  moreover
  have traces (composition) ⊆ traces (ioa 0 id2)
    — Finally we show trace inclusion
  proof −
    have ext (composition) = ext (ioa 0 id2)
      — First we show that they have the same external signature
      using same-input-sig and same-output-sig by simp
    moreover
    have is-ref-map f (composition) (ioa 0 id2)
      — Then we show that f-comp is a refinement mapping
    proof (auto simp only:is-ref-map-def)
      fix s1 s2
      assume 1:(s1,s2) ∈ ioa.start (composition)
      show f (s1,s2) ∈ ioa.start (ioa 0 id2)
      proof −
        have 2:ioa.start (ioa 0 id2) = start (0::nat) by simp
        have 3:ioa.start (composition)
          = start (0::nat) × start id1 by fastforce
        show ?thesis
          using 1 2 3 by simp
      qed
    next
      fix s1 s2 t1 t2 :: ('a,'b,'c)SLin-state and a :: ('a,'b,'c,'d)SLin-action
      assume reach:reachable (composition) (s1,s2)
      and trans:(s1,s2) −a−(composition)⟶ (t1,t2)
      define u where u = f (s1,s2)
      define u' where u' = f (t1,t2)
```

Lemmas and invariants

```
      have pendingReqs s2 ⊆ pendingReqs u
      proof −
        have P6 (s1,s2) using reach P6-invariant
          by (metis invariant-def)
        thus ?thesis
          by (force simp add:pendingReqs-def u-def)
      qed
      note lem1 = this
```

```
have initialized u by (auto simp add:u-def)
have P1 (s1,s2) and P1 (t1,t2) using reach P1-invariant
  trans invariant-def by (metis , metis reachable-n)
have P6 (s1,s2) and P6 (t1,t2) using reach P6-invariant
  trans invariant-def by (metis , metis reachable-n)
have P7 (s1,s2) and P7 (t1,t2) using reach P7-invariant
  trans invariant-def by (metis , metis reachable-n)
have P8 (s1,s2) and P8 (t1,t2) using reach P8-invariant
  trans invariant-def by (metis , metis reachable-n)
have P9 (s1,s2) and P9 (t1,t2) using reach P9-invariant
  trans invariant-def by (metis , metis reachable-n)
have P10 (s1,s2) and P10 (t1,t2) using reach P10-invariant
  trans invariant-def by (metis , metis reachable-n)
have P13 (s1,s2) and P13 (t1,t2) using reach P13-invariant
  trans invariant-def by (metis , metis reachable-n)
have P15 (s1,s2) and P15 (t1,t2) using reach P15-invariant
  trans invariant-def by (metis , metis reachable-n)
have P16 (s1,s2) and P16 (t1,t2) using reach P16-invariant
  trans invariant-def by (metis , metis reachable-n)
have P17 (s1,s2) and P17 (t1,t2) using reach P17-invariant
  trans invariant-def by (metis , metis reachable-n)
have P19 (s1,s2) and P19 (t1,t2) using reach P19-invariant
  trans invariant-def by (metis , metis reachable-n)
have P21 (s1,s2) and P21 (t1,t2) using reach P21-invariant
  trans invariant-def by (metis , metis reachable-n)
have P22 (s1,s2) and P22 (t1,t2) using reach P22-invariant
  trans invariant-def by (metis , metis reachable-n)
have P25 (s1,s2) and P25 (t1,t2) using reach P25-invariant
  trans invariant-def by (metis , metis reachable-n)
have P8a (s1,s2) and P8a (t1,t2) using reach P8a-invariant
  trans invariant-def by (metis , metis reachable-n)
have P23 (s1,s2) and P23 (t1,t2) using reach P23-invariant
  trans invariant-def by (metis , metis reachable-n)

show ∃ e . refines e (s1,s2) a (t1,t2) (ioa 0 id2) f
  using trans
proof (cases rule:trans-elim)
  case (Invoke1 i p c)
  let ?e = (u,[(a,u′)])
  have 1:is-exec-frag-of (ioa 0 id2) ?e
  proof −
    have 1:status s1 p = Ready and 2:t2 = s2
    and 3:t1 = s1⦇pending := (pending s1)(p := (p,c)),
      status := (status s1)(p := Pending)⦈
      using Invoke1(1) by auto
    have 4:status u p = Ready using 1 u-def by auto
    have 5:u′ = u⦇pending := (pending u)(p := (p,c)),
      status := (status u)(p := Pending)⦈
        using 2 3 u-def u′-def by auto
```

```
          have 6:Inv p c u u' using 4 5 by force
          show ?thesis using 6 Invoke1(3) ids by simp
        qed
        have 2:a ∈ ext (ioa 0 id2) and 3:trace (ioa.asig (ioa 0 id2)) ?e = [a]
             using Invoke1(2,3) ids by (auto simp add:trace-def schedule-def filter-act-def)
        show ?thesis using 1 2 3
          by (simp only:refines-def u-def u'-def)
            (metis fst-conv last-state.simps(2) snd-conv)
      next

        case (Invoke2 i p c)
        let ?e = (u,[(a,u')])
        have 1:is-exec-frag-of (ioa 0 id2) ?e
        proof −
          have 1:status s2 p = Ready and 2:t1 = s1
          and 3:t2 = s2⦇pending := (pending s2)(p := (p,c)),
            status := (status s2)(p := Pending)⦈
            using Invoke2(1) by auto
          have 4:status u p = Ready using 1 u-def ‹P6 (s1,s2)› by auto
          have 5:u' = u⦇pending := (pending u)(p := (p,c)),
            status := (status u)(p := Pending)⦈
              using 2 3 u-def u'-def ‹P6 (t1,t2)› by fastforce
          have 6:Inv p c u u' using 4 5 by force
          show ?thesis using 6 Invoke2(3) ids by simp
        qed
        have 2:a ∈ ext (ioa 0 id2)
        and 3:trace (ioa.asig (ioa 0 id2)) ?e = [a]
         using Invoke2(2,3) by (auto simp add:trace-def schedule-def filter-act-def)
        show ?thesis using 1 2 3
          by (simp only:refines-def u-def u'-def)
            (metis fst-conv last-state.simps(2) snd-conv)
      next

        case (Response2 i p ou)
        let ?e = (u,[(a,u')])
        have 1:is-exec-frag-of (ioa 0 id2) ?e
        proof −
          have 1:status s1 p = Aborted ∧ status t1 p = Aborted
          proof −
            show ?thesis using ‹P6 (s1,s2)› ‹P6 (t1,t2)›
              Response2(1) by force
          qed
          have 2:status u p = Pending ∧ initialized u
            using 1 Response2(1) u-def by auto
          have  3:u' = u⦇status := (status u)(p := Ready)⦈
            using 1 Response2(1) u-def u'-def
              by (cases u, cases u', auto)
          have 4:ou = γ (dstate u) (pending u p) ∧ contains (dstate u) (pending u
```

```
p)
          proof (cases dstate s2 = ⊥)
            case False
            thus ?thesis using 1 Response2(1) u-def by auto
          next
            case True
            have dstate s1 = dstate s2
            proof -
              have dstate s1 ≼ dstate s2
                using Response2(1) ‹P9 (s1,s2)› by auto
              with True show ?thesis by (metis antisym bot)
            qed
            thus ?thesis using 1 Response2(1) u-def by auto
          qed
          show ?thesis using 2 3 4 Response2(3) ids by auto
        qed
        have 2:a ∈ ext (ioa 0 id2)
        and 3:trace (ioa.asig (ioa 0 id2)) ?e = [a]
          using Response2(2,3) ids
            by (auto simp add:trace-def schedule-def filter-act-def)
        show ?thesis using 1 2 3
          by (simp only:refines-def u-def u′-def)
            (metis fst-conv last-state.simps(2) snd-conv)
      next

        case (Response1 i p ou)
        let ?e = (u,[(a,u′)])
        have 1:is-exec-frag-of (ioa 0 id2) ?e
        proof (cases dstate s2 = ⊥)
          case True
          have 1:status u p = Pending ∧ initialized u
            using Response1(1) u-def by auto
          have  2:u′ = u⦇status := (status u)(p := Ready)⦈
            using Response1(1) u-def u′-def
              by (cases u, cases u′, auto)
          have 3:ou = γ (dstate u) (pending u p)
            ∧ contains (dstate u) (pending u p)
            using Response1(1) True u-def by auto
          show ?thesis using 1 2 3 ‹initialized u› Response1(3) ids by auto
        next
          case False
          have 1:status u p = Pending ∧ initialized u
            using Response1(1) u-def by auto
          have  2:u′ = u⦇status := (status u)(p := Ready)⦈
            using Response1(1) u-def u′-def
              by (cases u, cases u′, auto)
          have 3:ou = γ (dstate u) (pending u p)
            and 4:contains (dstate u) (pending u p)
          proof -
```

```
      have 2:contains (dstate s1) (pending s1 p)
        using Response1(1) by auto
      show contains (dstate u) (pending u p)
      proof −
        have 3:dstate s1 ≼ dstate u
        proof −
          have initialized s2 using ‹P16 (s1,s2)› False
            by auto
          thus ?thesis using ‹P9 (s1,s2)› u-def False refl by simp
        qed
        have 4:pending s1 p = pending u p
          using u-def Response1(1) by force
        show ?thesis
          using 2 3 4 by (metis contains-star less-eq-def)
      qed
      have 4:γ (dstate s1) (pending s1 p) = γ (dstate u) (pending u p)
      proof −
        have 4:pending s1 p = pending u p
          using u-def Response1(1) by force
        obtain rs where 5:dstate u = dstate s1 ⋆ rs
          and 6:∀ r ∈ set rs . fst r ≠ p
        proof −
          have 7:dstate u = dstate s2 using u-def False by simp
          have 6:status s1 p = Pending
            ∧ contains (dstate s1) (pending s1 p)
              using Response1(1) by force
          have 8:initialized s2 using False ‹P16 (s1,s2)›
            by auto
          show ?thesis using that ‹P17 (s1,s2)› 6 8 7 by fastforce
        qed
        have 7:fst (pending s1 p) = p
          using Response1(1) ‹P1 (s1,s2)› by auto
        show ?thesis using 4 5 6 7 2 idem2-star by auto
      qed
      thus ou = γ (dstate u) (pending u p)
        using Response1(1) by simp
    qed
    thus ?thesis using 1 2 3 Response1(3) ids by auto
  qed
  have 2:a ∈ ext (ioa 0 id2)
  and 3:trace (ioa.asig (ioa 0 id2)) ?e = [a]
    using Response1(2,3) ids
      by (auto simp add:trace-def schedule-def filter-act-def)
  show ?thesis using 1 2 3
    by (simp only:refines-def u-def u′-def)
      (metis fst-conv last-state.simps(2) snd-conv)
next

  case (Reco2)
```

```
let ?e = (u,[(Linearize 0,u')])
have is-exec-frag-of (ioa 0 id2) ?e
proof -
  obtain rs where 1:rs ∈ pendingSeqs u
    and 2:dstate u' = dstate u ⋆ rs
    and 3:∀ av ∈ abortVals u . dstate u' ⪯ av
  proof -
    obtain rs where set rs ⊆ pendingReqs s1 ∪ pendingReqs s2
    and dstate t2 = dstate s1 ⋆ rs
    and ∀ av ∈ abortVals s2 . dstate t2 ⪯ av
    proof -
      obtain ivs rs where 3:ivs ⊆ initVals s2 and 4:ivs ≠ {}
      and 5:dstate t2 = ⨅ ivs ⋆ rs and 7:rs ∈ pendingSeqs s2
      and 6:∀ av ∈ abortVals s2 . dstate t2 ⪯ av
        using Reco2(1)
        by (auto simp add:safeInits-def initSets-def)
          (metis all-not-in-conv)
      obtain rs' where ⨅ ivs = dstate s1 ⋆ rs'
      and set rs' ⊆ pendingReqs s1
      proof -
        { fix iv
          assume 7:iv ∈ ivs
          have ∃ rs . set rs ⊆ pendingReqs s1
            ∧ iv = dstate s1 ⋆ rs
              using ‹P8 (s1,s2)› 7 3 by auto
                (metis mem-Collect-eq pendingSeqs-def rev-subsetD) }
        moreover have finite ivs using ‹P13 (s1,s2)› 3
            by (metis P13.simps rev-finite-subset)
        ultimately show ?thesis using that glb-common-set 4
          by metis
      qed
      hence dstate t2 = dstate s1 ⋆ (rs'@rs)
        ∧ set (rs'@rs) ⊆ pendingReqs s1 ∪ pendingReqs s2 using 5 7
          by (metis (lifting, no-types) Un-commute Un-mono
              exec-append mem-Collect-eq pendingSeqs-def set-append)
      thus ?thesis using that 6 by blast
    qed
    moreover
    have pendingReqs s1 ∪ pendingReqs s2 ⊆ pendingReqs u
    proof -
      note ‹pendingReqs s2 ⊆ pendingReqs u›
      moreover
      have pendingReqs s1 ⊆ pendingReqs u
        using Reco2(1) ‹P7 (s1,s2)›
          by (auto simp add:pendingReqs-def u-def)
      ultimately show ?thesis by auto
    qed
    moreover
    have abortVals u = abortVals s2 by (auto simp add:u-def)
```

```
      moreover
      have dstate u = dstate s1 using ‹P16 (s1,s2)›
        Reco2(1) u-def by force
      moreover
      have dstate u′ = dstate t2
        using Reco2(1) ‹P22 (t1,t2)› by (auto simp add:u′-def)
      ultimately show ?thesis using that
        by (auto simp add:pendingSeqs-def, blast)
    qed
    moreover
    have u′ = u⦇dstate := dstate u ⋆ rs⦈
      using 2 Reco2(1) u-def u′-def by force
    moreover
    note ‹initialized u›
    ultimately show ?thesis by auto
  qed
  moreover
  have a ∉ ext (ioa 0 id2)
  and trace (ioa.asig (ioa 0 id2)) ?e = []
    using Reco2(2) ids
      by (auto simp add:trace-def schedule-def filter-act-def)
  ultimately show ?thesis
    by (simp only:refines-def u-def u′-def)
      (metis fst-conv last-state.simps(2) snd-conv)
next

  case (Switch1 p c av)
  let ?e = (u,[])
  have is-exec-frag-of (ioa 0 id2) ?e by auto
  moreover
  have a ∉ ext (ioa 0 id2)
  and trace (ioa.asig (ioa 0 id2)) ?e = []
    using Switch1(2) ids
      by (auto simp add:trace-def schedule-def filter-act-def)
  moreover
  have u = u′ using Switch1(1) u-def u′-def by auto
  ultimately show ?thesis
    using refines-def[of ?e (s1,s2) a (t1,t2) ioa 0 id2 f]
      u-def u′-def by (metis last-state.simps(1) fst-conv)
next

  case Lin2
  let ?e = (u,[(Linearize 0,u′)])
  have is-exec-frag-of (ioa 0 id2) ?e
  proof −
    have u′ = u⦇dstate := dstate u′⦈ using Lin2(1)
      by (auto simp add:u-def u′-def)
    moreover
    note ‹initialized u›
```

```
    moreover
    obtain rs where dstate u′ = dstate u ⋆ rs
      and rs ∈ pendingSeqs u
      and ∀ av ∈ abortVals u . dstate u′ ≼ av
    proof −
      obtain rs where 1:dstate t2 = dstate s2 ⋆ rs
        and 2:rs ∈ pendingSeqs s2
        and 3:∀ av ∈ abortVals s2 . dstate t2 ≼ av
        using Lin2(1) by force
      have 4:rs ∈ pendingSeqs u
        using 2 and ‹pendingReqs s2 ⊆ pendingReqs u›
          by (metis mem-Collect-eq pendingSeqs-def subset-trans)
      have 5:dstate u′ = dstate u ⋆ rs
        and 6:∀ av ∈ abortVals u . dstate u′ ≼ av
      proof −
        have 7:dstate u = dstate s2 ∧ dstate u′ = dstate t2
          using ‹P22 (s1,s2)› and ‹P22 (t1,t2)› Lin2(1)
            by (auto simp add:u-def u′-def)
        show dstate u′ = dstate u ⋆ rs using 7 1 by auto
        show ∀ av ∈ abortVals u . dstate u′ ≼ av
        proof −
          have abortVals s2 = abortVals u by (auto simp add:u-def)
          thus ?thesis using 7 3 by simp
        qed
      qed
      show ?thesis using that 4 5 6 by auto
    qed
    ultimately show ?thesis by auto
  qed
  moreover
  have a ∉ ext (ioa 0 id2)
  and trace (ioa.asig (ioa 0 id2)) ?e = []
    using Lin2(2) ids
      by (auto simp add:trace-def schedule-def filter-act-def)
  ultimately show ?thesis
    by (simp only:refines-def u-def u′-def)
      (metis fst-conv last-state.simps(2) snd-conv)
next

  case Lin1
  have u′ = u⦇dstate := dstate u′⦈ using Lin1(1)
    by (auto simp add:u-def u′-def)
  show ?thesis
  proof (cases initialized s2)
    case False
    let ?e = (u,[(Linearize 0,u′)])
    have is-exec-frag-of (ioa 0 id2) ?e
    proof −
      note ‹u′ = u⦇dstate := dstate u′⦈›
```

```
  moreover
  note ‹initialized u›
  moreover
  obtain rs where dstate u′ = dstate u ⋆ rs
    and rs ∈ pendingSeqs u
    and ∀ av ∈ abortVals u . dstate u′ ≼ av
  proof −
    obtain rs where 1:dstate t1 = dstate s1 ⋆ rs
      and 2:rs ∈ pendingSeqs s1
      and 3:∀ av ∈ abortVals s1 . dstate t1 ≼ av
      using Lin1(1) by force
    have 5:pendingSeqs s1 ⊆ pendingSeqs u
      using False ‹P7 (s1,s2)›
        by (auto simp add:pendingReqs-def pendingSeqs-def u-def)
    have 6:dstate u = dstate s1 ∧ dstate u′ = dstate t1
      using ‹P16 (s1,s2)› False Lin1(1)
        by (auto simp add:u-def u′-def)
    have 4:∀ av ∈ abortVals u . dstate u′ ≼ av
    proof (cases abortVals u = {})
      case True
      thus ?thesis by auto
    next
      case False
      have dstate u′ = dstate t1 using 6 by auto
      moreover have abortVals u = abortVals t2
        using Lin1(1) by (auto simp add:u-def)
      moreover have dstate t1 ≼ ⨅(abortVals t1)
      proof −
        have abortVals t1 = abortVals s1 using Lin1(1) by auto
        moreover have abortVals t1 ≠ {} using False ‹P19 (t1,t2)›
          Lin1(1) by (simp add: u-def)
        ultimately show ?thesis using 3 ‹P13 (t1,t2)›
          by simp (metis boundedI)
      qed
      ultimately show ?thesis using ‹P21 (t1,t2)› 3
        by (metis P21.simps coboundedI2 orderE)
    qed
    show ?thesis using 1 2 3 4 5 6 that by auto
  qed
  ultimately show ?thesis by auto
qed
moreover
have a ∉ ext (ioa 0 id2)
and trace (ioa.asig (ioa 0 id2)) ?e = []
  using Lin1(2) ids
    by (auto simp add:trace-def schedule-def filter-act-def)
ultimately show ?thesis
  by (simp only:refines-def u-def u′-def)
    (metis fst-conv last-state.simps(2) snd-conv)
```

```
  next
    case True
    let ?e = (u,[])
    have is-exec-frag-of (ioa 0 id2) ?e by auto
    moreover
    have a ∉ ext (ioa 0 id2)
    and trace (ioa.asig (ioa 0 id2)) ?e = []
      using Lin1(2) ids
        by (auto simp add:trace-def schedule-def filter-act-def)
    moreover have last-state ?e = u′
    proof −
      have dstate u = dstate s2 ∧ dstate u′ = dstate t2
        using ‹P22 (s1,s2)› and ‹P22 (t1,t2)› and True and Lin1(1)
          by (auto simp add:u-def u′-def)
      thus ?thesis using Lin1(1) ‹u′ = u⦇dstate := dstate u′⦈›
        by simp
    qed
    ultimately show ?thesis
      using refines-def[of ?e (s1,s2) a (t1,t2) ioa 0 id2 f]
        by (simp only:u-def u′-def, auto)
  qed
next

  case (Switch2 p c av)
  let ?e = (u,[(a,u′)])
  have 1:is-exec-frag-of (ioa 0 id2) ?e
  proof −
    have 1:u′ = u⦇abortVals := (abortVals u) ∪ {av},
      status := (status u)(p := Aborted)⦈
      and 2:av ∈ safeAborts s2 and 3:status u p = Pending
      and 4:pending u p = (p,c)
    proof −
      have 1:t2 = s2⦇abortVals := (abortVals s2) ∪ {av},
        status := (status s2)(p := Aborted)⦈
        and 2:av ∈ safeAborts s2 and 3:s1 = t1
        and 4:status s2 p = Pending
          using Switch2(1) by auto
      show 5:status u p = Pending using ‹P6 (s1,s2)› 4
        by (auto simp add:u-def)
      have 6:status u′ p = Aborted using ‹P6 (t1,t2)› 1
        by (auto simp add:u′-def)
      show pending u p = (p,c) using ‹P6 (s1,s2)› 4 Switch2(1)
        by (auto simp add:u-def)
      show u′ = u⦇abortVals := (abortVals u) ∪ {av},
        status := (status u)(p := Aborted)⦈ using 1 3 5 6
          by (auto simp add:u-def u′-def)
      show av ∈ safeAborts s2 using 2 by assumption
    qed
    have 5:av ∈ safeAborts u
```

```
proof (cases initialized s2)
  case True
  hence 6:dstate u = dstate s2 using ‹P22 (s1,s2)›
    by (auto simp add:u-def)
  have (∃ rs ∈ pendingSeqs s2 . av = dstate s2 ⋆ rs)
    ∨ (dstate s2 ≼ av ∧ (∃ ivs ∈ initSets s2 .
      dstate s2 ≼ ⨅ivs ∧ (∃ rs ∈ pendingSeqs s2 . av = ⨅ivs ⋆ rs)))
  proof −
    have av ∈ initAborts s2
      using 2 and True by (auto simp add:safeAborts-def)
    thus ?thesis by (auto simp add:initAborts-def)
  qed
  thus ?thesis
  proof
    assume ∃ rs ∈ pendingSeqs s2 . av = dstate s2 ⋆ rs
    moreover note ‹initialized u›
    ultimately show ?thesis using ‹pendingReqs s2 ⊆ pendingReqs u› 6
      by (simp add:safeAborts-def initAborts-def)
        (metis less-eq-def mem-Collect-eq pendingSeqs-def
          sup.coboundedI2 sup.orderE)
  next
    assume 7:dstate s2 ≼ av ∧ (∃ ivs ∈ initSets s2 .
      dstate s2 ≼ ⨅ivs ∧ (∃ rs ∈ pendingSeqs s2 . av = ⨅ivs ⋆ rs))
    show ?thesis
    proof −
      have 8:dstate u ≼ av using 7 6 by auto
      obtain ivs rs' where 9:ivs ∈ initSets s2
        and 10:dstate s2 ≼ ⨅ivs
        and 11:rs' ∈ pendingSeqs s2 ∧ av = ⨅ivs ⋆ rs'
          using 7 by auto
      have 12:dstate u = dstate s2 using True ‹P22 (s1,s2)›
        by (auto simp add:u-def)
      moreover
      obtain rs where rs ∈ pendingSeqs u and ⨅ivs = dstate s2 ⋆ rs
        using ‹P25 (s1,s2)› True 9 10 by (auto simp add:u-def)
      ultimately have  av = dstate u ⋆ (rs@rs')
        and rs@rs' ∈ pendingSeqs u
        using 11 by (simp-all add:pendingSeqs-def)
          (metis exec-append, metis lem1 subset-trans)
      thus ?thesis using 8 ‹initialized u›
        by (auto simp add:safeAborts-def initAborts-def)
    qed
  qed
next
  case False
  with 2 have 0:av ∈ uninitAborts s2 by (auto simp add:safeAborts-def)
  show ?thesis
  proof −
    obtain ivs rs where 1:ivs ∈ initSets s2
```

```
                and 2:rs ∈pendingSeqs s2
                and 3:av = ⨅ ivs ⋆ rs
                using 0 by (auto simp add:uninitAborts-def)
              have 4:rs ∈ pendingSeqs u using lem1 2
                by (auto simp add:pendingSeqs-def)
              have 5:dstate u = dstate s1 using False ‹P10 (s1,s2)›
                by (auto simp add:u-def)
              obtain rs′ where 6:⨅ ivs = dstate s1 ⋆ rs′
                and 7:rs′ ∈ pendingSeqs s1
                  using 1 ‹P8a (s1,s2)› by auto
              have 8:rs′ ∈ pendingSeqs u using False ‹P23 (s1,s2)› 7
                by (auto simp add:u-def)
              have 9:av = dstate u ⋆ (rs′@rs) using 3 5 6
                by (metis exec-append)
              have 10:rs′@rs ∈ pendingSeqs u
                using 4 8 by (auto simp add:pendingSeqs-def)
              show ?thesis using 9 10 ‹initialized u›
                by (auto simp add:safeAborts-def initAborts-def less-eq-def)
            qed
          qed
          show ?thesis using 1 3 4 5 Switch2(2) by auto
        qed
        moreover
        have a ∈ ext (ioa 0 id2)
        and trace (ioa.asig (ioa 0 id2)) ?e = [a]
          using Switch2(2) ids
            by (auto simp add:trace-def schedule-def filter-act-def)
        ultimately show ?thesis
          by (simp only:refines-def u-def u′-def)
            (metis fst-conv last-state.simps(2) snd-conv)
      qed
    qed
    ultimately show ?thesis using ref-map-soundness by blast
  qed
  ultimately show ?thesis by (metis ioa-implements-def)
qed

end

end
```

8 The Consensus Data Type

```
theory Consensus
imports RDR
begin
```

This theory provides a model for the RDR locale, thus showing that the assumption of the RDR locale are consistent.

```
typedecl proc
typedecl val

locale Consensus
— To avoid name clashes
begin

fun δ::val option ⇒ (proc × val) ⇒ val option (infix ‹•› 65) where
  δ None r = Some (snd r)
| δ (Some v) r = Some v

fun γ::val option ⇒ (proc × val) ⇒ val where
  γ None r = snd r
| γ (Some v) r = v

interpretation pre-RDR δ γ None .
notation exec (infix ‹⋆› 65)
notation less-eq (infix ‹⪯› 50 )
notation None (‹⊥›)

lemma single-use:
  fixes r rs
  shows ⊥ ⋆ ([r]@rs) = Some (snd r)
proof (induct rs)
  case Nil
  thus ?case by simp
next
  case (Cons r rs)
  thus ?case by auto
qed

lemma bot: ∃ rs . s = ⊥ ⋆ rs
proof (cases s)
  case None
  hence s = ⊥ ⋆ [] by auto
  thus ?thesis by blast
next
  case (Some v)
  obtain r where ⊥ ⋆ [r] = Some v by force
  thus ?thesis using Some by metis
qed

lemma prec-eq-None-or-equal:
fixes s1 s2
assumes s1 ⪯ s2
shows s1 = None ∨ s1 = s2 using assms single-use
proof –
  { assume 1:s1 ≠ None and 2:s1 ≠ s2
    obtain r rs where 3:s1 = ⊥ ⋆ ([r]@rs) using bot using 1
```

```
      by (metis append-butlast-last-id pre-RDR.exec.simps(1))
    obtain rs' where 4:s2 = s1 ⋆ rs' using assms
      by (auto simp add:less-eq-def)
    have s2 = ⊥ ⋆ ([r]@(rs@rs')) using 3 4
      by (metis exec-append)
    hence s1 = s2 using 3
      by (metis single-use)
    with 2 have False by auto }
  thus ?thesis by blast
qed

interpretation RDR δ γ ⊥
proof (unfold-locales)
  fix s r
  assume contains s r
  show s · r = s
  proof –
    obtain rs where s = ⊥ ⋆ rs and rs ≠ []
      using ‹contains s r›
      by (auto simp add:contains-def, force)
    thus ?thesis
    by (metis δ.simps(2) rev-exhaust single-use)
  qed
next
  fix s and r r' :: proc × val
  assume 1:fst r ≠ fst r'
  thus γ s r = γ ((s · r) · r') r
    by (metis δ.simps γ.simps not-Some-eq)
next
  fix s1 s2
  assume s1 ≼ s2 ∧ s2 ≼ s1
  thus s1 = s2 by (metis prec-eq-None-or-equal)
next
  fix s1 s2
  show ∃ s . is-glb s s1 s2
  by (simp add:is-glb-def is-lb-def)
    (metis bot pre-RDR.less-eq-def prec-eq-None-or-equal)
next
  fix s
  show ⊥ ≼ s
  by (metis bot pre-RDR.less-eq-def)
next
  fix s1 s2 s3 rs
  assume s1 ≼ s2 and s2 ≼ s3 and s3 = s1 ⋆ rs
  thus ∃ rs' rs'' . s2 = s1 ⋆ rs' ∧ s3 = s2 ⋆ rs''
    ∧ set rs' ⊆ set rs ∧ set rs'' ⊆ set rs
  by (metis Consensus.prec-eq-None-or-equal
      in-set-insert insert-Nil list.distinct(1)
        pre-RDR.exec.simps(1) subsetI)
```

qed

end

end

9 Conclusion

In this document we have defined the SLin I/O-automaton (a shorthand for Speculative Linearizability) and we have proved that the composition of two instances of the SLin I/O-automaton behaves like a single instance of the SLin I/O-automaton. This theorem justifies the compositional proof technique presented in [4].

References

[1] R. Guerraoui, V. Kuncak, and G. Losa. Speculative linearizability. In J. Vitek, H. Lin, and F. Tip, editors, *PLDI*, pages 55–66. ACM, 2012.

[2] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[3] L. Lamport and K. Marzullo. The part-time parliament. *ACM Transactions on Computer Systems*, 16:133–169, 1998.

[4] G. Losa. *Modularity in the design of robust distributed algorithms*. PhD thesis, EPFL, 2014.

[5] N. A. Lynch and M. R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2:219–246, 1989.